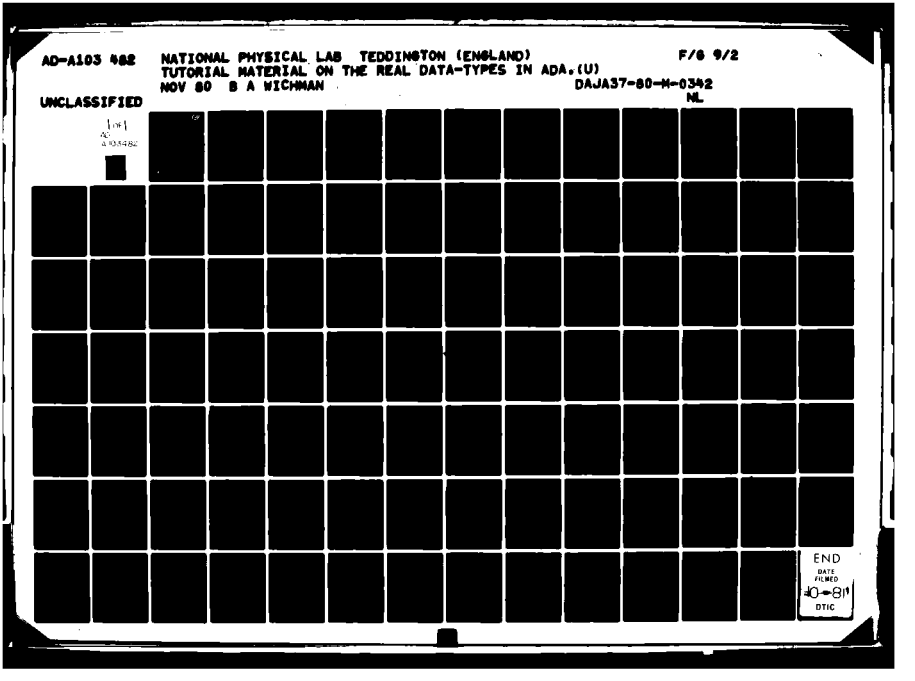

AD-A103 482
NATIONAL PHYSICAL LAB TEDDINGTON (ENGLAND)
F/G 9/2
TUTORIAL MATERIAL ON THE REAL DATA-TYPES IN ADA.(U)
NOV 80 B A WICHMAN
DAJA37-80-M-0342
UNCLASSIFIED
NL
1 OF 1
AD A103482
END
DATE FILMED
10-81
DTIC

TUTORIAL MATERIAL ON THE REAL DATA-TYPES IN ADA

Final Technical Report

by

B A Wichmann

November 1980

United States Army

EUROPEAN RESEARCH OFFICE
London, England

CONTRACT NUMBER DAJA37-80-M-0342

National Physical Laboratory
Teddington, Middlesex, TW11 OLW, UK

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A103 482	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Tutorial Material on the Real Data-Types in Ada.	5. TYPE OF REPORT & PERIOD COVERED
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Brian A. Wichmann	8. CONTRACT OR GRANT NUMBER(s) DAJA37-80-M-0342
9. PERFORMING ORGANIZATION NAME AND ADDRESS National Physical Laboratory Teddington, Middlesex, TW11 0LW, UK	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army European Resaerch Office, London, England	12. REPORT DATE November, 1980
	13. NUMBER OF PAGES 38 pages
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report)
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this Document is Unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada, real arithmetic, floating point, fixed point, arithmetic types, numeric types, approximate computation, Blue's algorithm

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Ada programming language introduces a number of novel features in the area of numerics. The purpose of this report is to present these features to the programmer who is familiar with numerical computation but not with Ada. The report is designed to be presented as a lecture with the aid of viewgraphs (drafts of these are included). However, the material can be read in conventional fashion.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Abstract. The Ada programming language introduces a number of novel features in the area of numerics. The purpose of this report is to present these features to a programmer who is familar with numerical computation but not with Ada. The report is designed to be presented as a lecture with the aid of viewgraphs (drafts of these are included). However, the material can be read in the conventional fashion.

Comments. The author would appreciate comments on the text so that any subsequent revision can incorporate improvements.

Acknowledgment. Mr G T Anthony and Miss H M Williams of NPL have reviewed this report which has resulted in substantial improvements in its style and content.

CONTENTS

Tutorial material on the real data-types in Ada

B A Wichmann, National Physical Laboratory

Note: These notes are designed as material to be presented with a set of viewgraphs. The complete material can be presented in about four hours assuming only a limited knowledge of Ada beforehand. The viewgraphs are reproduced at the end of these notes, and are referenced in the text by numbers in the right hand margin.

(1)

1. Fixed and Floating point

The real data types in Ada are for approximate computation. The majority of physical quantities are necessarily approximate because of the inherent errors involved in their observation. Such quantities are therefore naturally handled by means of the real data types in Ada. The purpose of these notes is to explain the facilities in Ada so that the programmer can use the language reliably and in a manner appropriate to the job in hand.

The real data types in Ada are divided into two classes - fixed point and floating point. There can be any number of fixed point and floating point data types in a program. It is convenient to have an intuitive view as to what fixed point and floating point means. Thinking in decimal, fixed point means a fixed number of places before the decimal point and a fixed number after:

±d.dd or ±ddd.d or ±ddd.

whereas floating point means that there are a fixed number of significant digits and an exponent ("scientific notation" of calculators): (2)

±d.ddE±dd or ±d.dE±d or ±d.dddE±dd

where the integer after the E gives the decimal exponent.

The data type thus determines how values are stored since any one type will have the same format. With a fixed point data type with the format

±d.dd

a half is stored as +0.50 and one third as +0.33 which is, of course, in error to a small extent. The fact that computed values and even constants cannot be stored exactly is the reason why real

data types are said to be approximate. Note that with this data type values of magnitude less than 0.005 will be represented as zero (assuming rounding is performed).

Now consider an example of a floating point data type with the format: ③

±d.ddE±d

Then

100.0 is stored as +1.00E+2

One might think it could also be stored as +0.10E+3 but this is not permitted because values are `normalized'. The importance of normalisation is easy to appreciate when considering storing the value 101.0 with the same format. This is

+1.01E+2

whereas putting an initial zero would lose the final 1 giving a one per cent error. Note that floating point values have a roughly constant relative error whereas fixed point quantities have a constant maximum absolute error.

In Ada, data types are distinguished by their names, not just their formats. Hence two data types having identical formats are distinct. This means that data types should be given names to reflect the logical properties rather than their formats. If two sensors read temperature and distance, then they should be given distinct data types DEGREES and FEET rather that one type just because the range of values and accuracy requires an identical format.

Note that the fixed point data types in Ada have formats which are not quite the same as those used by calculators. With a calculator, dividing 1.23 by ten gives .123, but with the format +d.dd, the division will yield +0.12. To do the division accurately in Ada, then the result must be stored in a type with the format +.ddd. Clearly, the reduced flexibility of the Ada fixed point means that it is very easy to lose accuracy in performing computations. Losses also arise with floating point computations but they are less marked due to the automatic normalization. For this reason, most programmers would prefer to use floating point, which is of course, why modern scientific computations use this mode. As a rough estimate, one should expect an algorithm to be three times more expensive to program in fixed point. The reason for using fixed point is usually the absence of floating point on a particular machine or because the digitised signal input is in fixed point. ④

The description given above using decimal formats is merely to illustrate the general nature of Ada data types. In fact, Ada

defines the data formats in binary since this is almost universal for modern computers. To give a more accurate description, we first need some notation from the Ada language.

2. Notation for literals

We are only concerned with numeric literals. These can either be for integers or real values. Real values are distinguished by the presence of a decimal point. If a real value is required by a particular context in the language, then an integer is not permitted. In other words, if the real value one is required 1.0 must be written and 1 will not be sufficient. This means that it is always easy to see if approximate computation is being performed, even with parameters to a procedure because literal values will have a decimal point.

Decimal integer values are written in the conventional manner. Spaces may not appear within the digits of the value, but an underscore can be used instead. This is very convenient with large values since the thousands or millions can be separated to aid the eye. (5)

Examples: 1 01_234_567

The following are not valid

1_ 2. 1__234

Large integer values can conveniently use the exponent notation. For instance, six million can be written as:

6_000_000 or 6E6 or 6_000E+3 etc.

An implementation may limit the size of literals which can be handled, but such limits are likely to be quite large. The line length also restricts the magnitude of literals.

Real literals can be written in the conventional decimal notation with a decimal point. An exponent can optionally be used. For instance, the following all represents the same value:

3.14 0.314E+1 314.0E-2 03.1_4000

The accuracy with which a literal value is stored in the program is determined by the context and not by the way in which the literal value is written in the program. Hence merely writing 20 decimal digits does not imply that the value will be stored with that accuracy. The accuracy will depend upon the types used in the computations containing the literal.

Both integer and real literals can be written using bases other than ten. One reason for this facility is that some machine

properties are specified by the manufacturer in octal or hexadecimal and hence this notation is the logical one to use in these contexts. An additional reason for permitting other bases for real literals will soon be apparent. The bases which Ada allows are those from 2 to 16. Base 16 uses a notation similar to that of hexadecimal on the IBM computers and in consequence A stands for 10, B for 11, C for 12, D for 13, E for 14 and F for 15. The base is determined by a decimal value before a sharp character which brackets the based number sequence. Note that the base and the exponent are written in decimal and in consequence the A-F characters when used as a digit, can only appear between the pair of sharp characters. For example:

2#101# means 4 + 1=5 with a base of 2

4#101# means $4^2 + 1 = 17$ (6)

16#FF# means 15 * 16 + 15 = 255

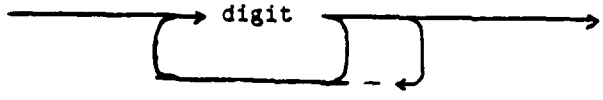
The notation can be used both with exponents and with a point for real literals.

Hence 4#101#E2 means 4#10100# = $4^4 + 4^2$ = 256+16 = 272

Writing and reading values in other bases requires care since we tend to think in decimal. This is especially true with real values.

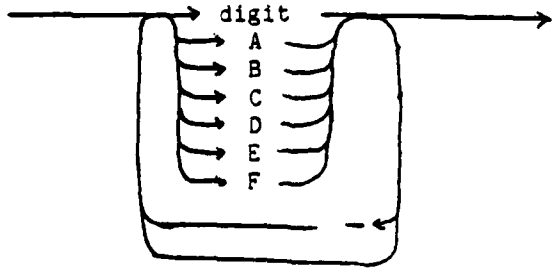
The syntax of numeric literals is most easily portrayed by means of syntax diagrams. The arrowed lines are followed according to the syntax units being analysed. For instance, an integer with interleaved understores permitted is given by the diagram

integer:



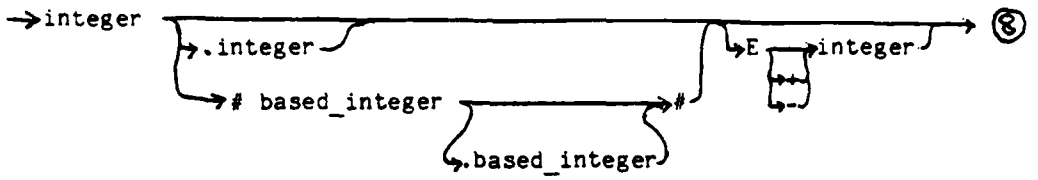
The box for digit can also be given by a diagram with just ten alternatives for each of the digits 0 to 9. Similarly, one has (7)

based_integer:



Now the syntax diagram for numeric literals can be given using the diagrams for integer and based_integer

numeric_literal:



(8)

One further language facility needs to be given because of its convenience in explaining the language later in this material. This is number declarations. Both real and integer literal values can be given an identifier in a number declaration. For instance

```
PI: constant := 3.14159_26535;
MAX_LINE_LENGTH: constant : = 96;
```

Within the Ada program where these identifiers can be used, the use of the identifier is equivalent to writing the literal value. Such number declarations can be used to separate out key numerical values.

Exercises

Write the following based number values in decimal:

16#FF# 4#1.01#E2 3#0.1# 8#0.1# 16#0.8# (9)

What value is 16#0.99999# just a bit less than?

What is wrong with the following literals?

3._14 4#_0.1#2 16#FF#E-1 8#0.9#

3. A model of approximate computation

Ada defines the properties that the approximate computation of real arithmetic must satisfy. Because the real arithmetic is implemented on machines with very different underlying hardware, the definition is permissive. In other words, the properties must be satisfied, but this can be achieved in a number of different ways. A particular real data type definition specifies an accuracy that must be met. An implementation is free to provide greater accuracy than that specified. This is essential because a machine can usually only conveniently implement a small range of different accuracies. The problem is to define the properties so that

different implementations are possible and yet make the properties good enough to meet the demands of the numerical analyst. The method used is based upon the work of W S Brown from Bell Laboratories on floating point [2]. There are differences between Brown's work and the definition of Ada because of the different objectives - Brown was interested in providing a model of actual hardware whereas with Ada a machine independent language definition is required. Ada also handles fixed point. (10)

Ada assumes that the arithmetic facilities are provided using binary. There are a few additional complexities with fixed point, so let us start by considering floating point. Floating point computation involves storing values with a sign, a mantissa and a signed exponent. The difficulty is that we do not wish to say how long the mantissa will be, nor the actual range of the exponent since this will depend upon the particular hardware in use. Hence we say that the mantissa must be at least so long, and the exponent range must be at least so long.

With a particular mantissa length and exponent range guaranteed, certain values are capable of being stored exactly. As an example, assume that the mantissa length is 4 hexadecimal places (corresponding to 16 binary places). Then

```
          16#0.8000#      = 0.5 is stored exactly
as is     16#0.F000#      = 15.0/16  = 0.9375
 and      16#0.FFFF#E4    = #FFFF.0# = 65535.0
```

With such a data type, these values are handled exactly in the sense that if one assigns a value to a variable, then one can test for equality and obtain the expected result. These values are called model numbers. Equality and inequality of model numbers have the characteristics of the exact values. However, an implementation will typically have values which are not model numbers and almost all the difficulties of real arithmetic are due to these values. Given one of these additional numbers it is usually bounded by a model interval. For instance, with the above data type

```
16# 0.ABCD4#      is bounded by
16# 0.ABCD#  and  16#0.ABCE#
```

and 0.1 which is 16#0.1999999...# is therefore bounded by

```
16#0.1999#  and  16#0.199A#
```

Literal values in an Ada program must be converted by the compiler to values within these bounds inclusively. Hence these model intervals perform a vital role in defining the errors that can arise in a computation. This role is extended to operations as follows. Given two operands A and B and an operation op, then we want to bound A op B. Corresponding to A and B, there are model intervals. The operation is then applied to the two intervals. The (11)

resulting set of values is then widened, if necessary, to a further model interval. This model interval bounds the machine computed value of A op B. This might seem complicated and indirect, but it has a number of simple consequences. For instance, the model interval for a model number is just the model number. Hence if the operands are both model numbers and the correct, mathematical result is also a model number, then the machine result must be the exact, correct result. (12)

Exactly the same logic of model numbers, model intervals and the calculation of model intervals which bounds the result of an operation applies to fixed point as well. The difference between fixed point and floating point lies in the model numbers themselves. There are some additional complexities which arise in the case when a computed result lies outside the range of model numbers.

Exercises

Given a floating point type which has model numbers with 4 hexadecimal places, what is

(a) the next model number above 1.0?
(b) the next model number below 1.0?
(c) the ratio of ((a) - 1.0)/(1.0 - (b))?

What rational numbers are not represented exactly in Ada with any accuracy using floating point?

4. Floating Point Data types

Ada allows the programmer to specify the minimal accuracy of a real data type. For floating point this specification is an integer giving the number of decimal digits of significance in stored values. This method of specification is used because of its strong intuitive appeal in spite of the fact that the detailed semantics of floating point uses binary.

The number of decimal digits determines the model numbers of the type. Since a binary radix is used, the floating point model numbers consist of

```
sign * binary_mantissa * (2.0 ** exponent)
```

(13)

where the mantissa length and the exponent range must be determined from the number of decimal digits. There is an obvious relationship between a binary mantissa and the corresponding decimal one. For D decimal digits one needs more than D*log(10)/log(2) binary places to give at least the same accuracy. Hence Ada defines the mantissa length to be the next integer greater than D*log(10)/log(2).

Unfortunately, there is no obvious natural value for the exponent range. In fact, the exponent range is independent and logically should be separately specified by the programmer. Such a detailed specification would not be useful since actual hardware does not permit an independent choice of both parameters. Also, to be convenient to the ordinary user, default values are needed for the range which would again be arbitrary. Some algorithms do require a reasonable range in relation to the mantissa and from a study of existing machines, the value has been set of -4*B .. 4*B where B is the number of binary places in the mantissa.

Let us now consider an example of a minimal working accuracy of five decimal digits. This requires at least 5 * 3.32 = 16.6 binary places. Hence the length of the binary mantissa is taken as 17 places. The binary exponent range is -68 .. 68. Hence we have

```
smallest model number greater than zero      = 2#0.1#E-68
                     = 2.0**(-69) about 1.69E-21

largest model number   = 2#0.1111111111111111#E68

                     = 2.0**68 - 2.0**51 about 2.95E20

The next model number greater than 1.0
                     = 2#0.1000000000000001#E1
                     = 1.0 + 2.0 **(-16)
```

(14)

Such a floating point type is defined by the declaration

```
type F is digits 5;
```

Having declared such a type, it is clearly convenient to be able to access the basic constants associated with it. By this means, algorithms can be written where the accuracy is isolated to the single type declaration. These constants are called attributes of the type and, in this case, they are predefined by the language definition.

The predefined attributes are written as the type identifier (F) a prime (') and then the name of the attribute. The attributes for a floating point type which are related to the model numbers are:

F'DIGITS: the value of the expression after `digits' in the type declaration, and hence 5 in this case,

F'MANTISSA: the binary length of the mantissa and hence 17 in this case,

F'EMAX: the maximum value of the exponent which is 68 in this case and is always 4*F'MANTISSA

F'SMALL: the smallest positive model number, which is about 1.69E-21 in this case. Its value is

always 2.0**(-F'EMAX-1).

F'LARGE: the largest model number, which is about 2.95E20 in this case. Its value is always 2.0**F'EMAX*(1.0-2.0**(-F'MANTISSA)).

F'EPSILON: the absolute value of the difference between 1.0 and the next model number above 1.0. The value in this case is about 1.52E-5 or in general 2.0**(-F'MANTISSA+1)

Of course, because of the relationship between these values, there is little logical need for them all. In practice, however, they are needed for program clarity. F'MANTISSA and F'EMAX give the basic properties of the model numbers whereas in actual programming the values F'SMALL, F'LARGE and F'EPSILON are usually needed.

Consider the problem of determining the errors in a computation. A literal value such as 0.1 cannot be stored exactly since it has a recurring binary representation. What is the error involved? In handling binary values, it is convenient to use hexadecimal otherwise the based numbers are rather long to write. We have (15)

0.1 = 16#0.19999...#

With the type F we have 17 binary places and hence the value 0.1 is bounded by the model interval

16#0.19999#..16#0.1999A#

The difference is 16#0.00001# = 16#0.1#E-4 = 16.0**(-5)
= F'EPSILON/16

The relative error is thus less than or equal to 9.54 E-6 in this case.

In general, it is easy to see that the relative error depends upon the relationship between the value and the powers of 2. For instance, a value just greater than one has a relative error of F'EPSILON (the definition of the value) whereas a value of just less than 1.0 has half that relative error. In practice, the actual values of constants are not so important, and in any case cannot be used for variables and hence the general rule is

lowest possible machine value representing the true value
= (1.0-F'EPSILON)*true value
highest possible machine value representing the true value
= (1.0+F'EPSILON)*true value

These are the relationships used for classical error analysis, combined of course, with corresponding relationships involving the numerical operations. Note that constants may be converted by

rounding implying half the maximum relative error. This paper does not aim to teach classical error analysis. The formal proof of the inequalities of classical error analysis from the Ada model number definition is given in References [1,2].

Classical error analysis is satisfactory provided the values computed are either zero or lie in the ranges F'SMALL .. F'LARGE and -F'LARGE .. F'SMALL. Consider the computation of a value (16) smaller in magnitude than F'SMALL, or even such a value written in a program. Then the value in the machine will be in interval -F'SMALL .. 0.0 or 0.0 ..F'SMALL according to the sign of the value. This implies that all precision could be lost. For instance, the actual machine may only handle model numbers and round literal values. Hence values greater than 0.0 and less then F'SMALL/2 will be converted to zero. A compiler could warn the programmer of such a conversion of a non-zero value to zero, but there would be little reason to do so since the same values calculated dynamically would lead to zero without warning. Hence the programmer needs to beware of this condition called underflow, if an algorithm requires the accurate computation of small values.

As an example of underflow, consider the computation of the length of the hypotenuse of a right angled triangle:

```
X := SQRT(A**2 + B**2);
```

It might seem reasonable that if A or B >= F'SMALL then X >= F'SMALL. However, F'SMALL**2 may underflow to 0.0, giving X=0.0 if both values are small. Hence if the specification of this calculation requires that non-zero values of A or B gives a non-zero value for X, then one must take this into account by writing (for instance):

```
SM: constant F := 2.0**(-F'EMAX/2);      --EMAX is even
-- calculate A and B
if ABS(A) < SM and ABS(B) < SM then
     A := A/SM;
     B := B/SM;
     X := SQRT(A**2 + B**2) * SM;
elsif
     -- other case
end if; -- (*1)
```

Note that the use of powers of two for scaling reduces the potential errors to a minimum.

Ada does not require that there are no machine values between 0.0 and F'SMALL. On a particular machine, such values could be present making the cautious code above less necessary. The programmer is

---

(*1) This example is merely an illustration, see section 10 for a realistic example.

strongly advised to take the precautions for underflow illustrated above because the algorithm will then be portable.

The problem of overflow, that is, when computed values or constants are greater than F'LARGE, is more severe. Clearly, there must be some limit to values that a machine can handle and beyond that limit it is, in general, unreasonable to replace the true value by a single value. Ada only requires that values upto F'LARGE are handled correctly. A machine can, and often does, provide further values. The implemented range for any Ada scalar type is F'FIRST .. F'LAST. When the implemented range of values is exceeded, most machines provide an indication of this fact. In Ada, this is signalled by means of the NUMERIC_ERROR exception, for computed values. If a literal value exceeds the implemented range, then the CONSTRAINT_ERROR exception is raised. With underflow, the computation proceeds in spite of obtaining potentially meaningless results, but with overflow an exception could lead to the termination of the computation. Hence the specification of a numeric computation should indicate if these exceptions can arise. The specification of a routine should indicate which of the following three cases hold with respect to the NUMERIC_ERROR and CONSTRAINT_ERROR exceptions:

(17)

(18)

(a) The routine has been written so as to avoid raising the exceptions.

(b) Local handlers have been written for the exceptions so that these exceptions cannot be propagated to the caller.

(c) The exceptions can indeed arise from a call of the routine (the conditions should be stated).

Consider now the computation

```
X := SQRT(A**2 + B**2);
```

but this time considering the question of overflow. The safest method is to avoid overflow by testing the values of A and B in a similar method of that used for underflow:

```
SL: constant F := 2.0**(F'EMAX/2-1);
-- calculate A and B
if ABS(A) > SL or ABS(B) > SL then
   A := A/SL;
   B := B/SL;
   X := SQRT(A**2 + B**2) * SL;
elsif
   -- other cases
end if; (*1)
```

An alternative strategy is to write a handler for the NUMERIC_ERROR exception and only in this case, scale for a large value. This is not to be recommended in general because it is machine dependent. The raising of the NUMERIC_ERROR exception is not guaranteed and indeed, on machines which allow computation with values representing infinity, the exception might never be raised.

A user can declare subtypes of a type (or subtype). Unlike a type, a subtype is potentially dynamic in its characteristics. Consider

```
type F is digits 5;
X:F := F(READ_FROM_DEVICE);
subtype TF is range 0.0 ..X;
```

Then the range of values that the subtype TF can have may vary from one execution of these declarations to another. On the other hand, the properties of F remain the same since the expression after `digits' is a static integer expression.

Subtypes of real types have both advantages and disadvantages in Ada. Obviously, it is useful to place bounds on values and have these bounds checked by the system as both a documentation aid and also to improve the reliability of the software. Unfortunately, the checking overhead on every assignment to variables of subtype TF is not insignificant. The check is necessary since the program is required to raise the exception CONSTRAINT_ERROR if the range is violated. The programmer can suppress the checking by means of a pragma, but this defeats the object of the facility. Hence subtypes with a real range constraint must be used with care.

Subtypes can also be used to indicate a need for less accuracy than that specified by the type definition. For instance:

```
subtype SF is F digits 4;
```

or just against an object

(*1) Again, this example is illustrative only, and section 10 gives a realistic example.

```
Y: F digits 4;
```

For a subtype, the model numbers are reduced by a corresponding reduction in the mantissa length, while keeping the exponent range the same. Hence this would mean a binary mantissa length of 14 places (3 less than F). This means that SF'LARGE is only a very small amount less than F'LARGE corresponding to losing three 1's at the least significant end of the binary mantissa. Note that SF'SMALL = F'SMALL. (19)

Since compilers must handle objects of a subtype in effectively the same way as objects of the type, it is unlikely that compilers can take much advantage of the reduced precision of a subtype. Hence the advantages of subtypes just giving an accuracy constraint are minimal. Since there is no checking for accuracy constraints at run-time, there is no run-time penalty.

Exercises

If F'DIGITS = 2*G'DIGITS, does F'MANTISSA = 2*G'MANTISSA? (20)

What is the largest positive value X:F such that X does not overflow and 1.0/X does not underflow?

5. The predefined floating point operations

For every floating point type, a conventional set of predefined operations are available as follows:

single operand	+	no operation
	-	change sign
two operands	*	multiplication
(of the same floating point type)	/	division
	+	addition
	-	subtraction
single parameter	ABS( )	absolute value

(21)

Each of these operations yields a result which is of the same type as the operands. The description of the error bounds and the circumstances under which the exception NUMERIC_ERROR can occur can now be given in detail (see 4.5.8 of manual), by means of the following steps:

1. For each operand, a model interval of the appropriate type or subtype is obtained.

2. The mathematical operation is performed on the model intervals, obtaining a new interval.

3. The interval from the last step is expanded, if necessary, to a model interval.

The model interval obtained from this last step bounds the accuracy of the operation.

Consider the computation of X/Y; X,Y:F and X=15.0 and Y=3.0. Both X and Y are model numbers (as are all small integers). Hence the two model intervals obtained from step 1 are just the two single values. Step two yields the mathematical result 5.0. Now step 3 gives the model interval consisting of this single value, since 5.0 is also a model number of type F. One can clearly see from this that computations involving small integer values and giving small integer values are exact.

Consider now a slightly more realistic example of X*Y, X,Y:F and X = 0.1 and Y = 10.0

Then X is in model interval 16#0.19999#..16#0.1999A#
and Y is the model number 16#0.A#E1

Step 2 then gives the interval 16#0.FFFFA#..16#1.00004#
Step 3 then gives the model interval 16#0.FFFF8#..16#1.0001#

If the programmer had written 0.1*Y in his program, then 0.1 is converted to the type F by the compiler and hence the same error analysis applies. Note that the resulting bounds are approximately symmetric about the correct result, although some actual machines may produce results with these bounds but with a bias. (22)

Consider another example of X+Y, X,Y:F with X=1.0 and Y=F'SMALL. Then the interval at step 2 is the single value 1.0 + F'SMALL but this is widened to the model interval 1.0 .. 1.0 + F'EPSILON. This analysis assumes that F'SMALL < F'EPSILON which is a consequence of fixing the exponent range in relation to the mantissa length.

One situation has not been detailed. In steps 1 and 3 above, it may be impossible to form a model interval because a value exceeds F'LARGE in absolute value. In this case, the interval is said to overflow. When this happens, the NUMERIC_ERROR exception may be raised. It need not be raised because the machine can handle larger values adequately or because no indication is given by the hardware. Because these different circumstances cannot be distinguished, portable software cannot rely upon the NUMERIC_ERROR exception.

One other operation is available for floating point which is irregular since the operands are of different types. This is the exponentiation operator written as **. The left hand operand is any floating point type and the right hand operand is any integer type. The result is of the same type as the left hand operand. The operation gives the result of repeatedly multiplying the left hand operand by itself for a positive exponent. The number of

multiplications being one less than the value of the right operand. A negative exponent gives the inverse of the positive expoent value. Hence:

```
X **2      is equivalent to X * X
X **(-2)          "         1.0/(X*X)
X **1             "         X
X ** 0            "         1.0
```

(23)

Hence the semantics of this operation are defined in terms of the multiplications involved. The compiler can reduce the number of multiplications by calculating X**4 as (X*X)*(X*X) rather than (X*X)*X)*X. This gives a faster computation for large values of the exponent but does not give (in general) more accuracy.

The remaining operations on floating point operands are more regular than ** but give a BOOLEAN result. These are the relational operators. All six relational operators are available although they must be used with caution, as we shall see.

In comparing two values, everything is straightforward if the two values are not approximately equal and both are in range (ie between - F'LARGE and F'LARGE). However, if the two values are nearly equal, one has a potential problem. Under such circumstances, the result will depend upon the actual accuracy of the hardware. The precise formulation of this again depends upon the use of model intervals as follows:

Firstly, the appropriate model intervals are constructed for each operand as in the case of the other operations. Then one of five cases determines the result:

(a) The intervals are disjoint: the mathematical result is obtained
(b) Each interval is the same single model number: the mathematical result is obtained (24)
(c) The two intervals intersect in a single model number: either the exact result is obtained or that of comparing one operand with itself
(d) The intervals have more than one number in common: the result is implementation dependent.
(e) One of the two intervals overflows: the result is again implementation dependent, but the NUMERIC_ERROR exception can be raised (although it need not).

These cases are easily illustrated by means of a table with type F of five digits again.

X	op	Y	case	result
0.1		10.1	(a)	mathematical result
F'SMALL		F'SMALL	(b)	mathematical result
0.1		0.1+F'EPSILON/8	(c)	(Intersect at 16#0.1999A#) mathematical result or 0.1 op 0.1 (=Y op Y)
0.1		0.1+F'SMALL	(d)	implementation defined
F'LARGE		-F'LARGE+1.0	(e)	implementation defined or NUMERIC_ERROR
0.1		0.1	(d)	implementation defined (*1)

Exercises

With A, B, C:FLOAT;

(25)

Does

```
(A + B) + C = A + (B + C)?
A + B = B + A?
```

What is wrong with the following?

```
A + 12
24*B
C**2.0?
```

6. Derivation from the hardware types for floating point

As explained so far, it would appear that an implementation would have to provide a large number of distinct types for digits N, N=1..30 (say). However, as is well known, machines typically have only one or two hardware types. We would appear to have a problem. However, as defined, an accuracy of N digits can be implemented with a hardware type having N or more digits of accuracy. Hence, given a machine with two hardware types of 10 and 20 digits accuracy, all the types of accuracy <= 10 would be handled with 10 digits, and the ones with more than 10 and less than or equal to

---

(*1) It might seem odd that 0.1=0.1 is not necessarily true. The reason is that many machines perform calculations with more accuracy than results can be stored in main memory. This is the so-called overlength accumulator. Hence 1.0/10.0 would give more accuracy than 0.1 stored in main memory, giving the unexpected false to 1.0/10.0 = 0.1.

20 digits with 20 digits of accuracy. The vital fact which permits this is that the model numbers for accuracy of digits N are model numbers for all larger accuracies. (26)

The hardware types have conventional names, namely SHORT_FLOAT, FLOAT and LONG_FLOAT. Of course, if there are only two hardware types, the names actually in use will depend upon the implementation. A valid Ada system could have no such types if the target hardware provides no approximate facilities. Assuming that floating point is provided, then type FLOAT should be available. Hence, if one is not concerned with control of accuracy for small amounts of code, then one can just use the type FLOAT. Direct use of the hardware types is not to be recommended since it is clearly machine dependent. However, if it is necessary to implement basic software effectively to augment the hardware, then such machine dependence is needed. Note that the attributes of FLOAT (SHORT_FLOAT and LONG_FLOAT) characterise the machine.

Given

```
type F is digits D;
```

such that F is implemented by the hardware type FLOAT, we say that F is derived from FLOAT which is written in full as

```
type F is new FLOAT digits D;
```

The full form is not appropriate in most cases, since on another system F could be implemented by SHORT_FLOAT. Hence the short form of declaration is to be preferred to increase portability. Even with the long form, F'DIGITS = D and this is not necessarily equal to FLOAT'DIGITS. Given either form of declaration, it is occasionally necessary to access the characteristics of the implemented type. This can be done by means of the notation (27)

```
F'BASE'DIGITS meaning FLOAT'DIGITS etc.
```

The advantage of the 'BASE notation is that it is possible to exploit the additional fortuitous accuracy provided by the implementation. Consider for instance the summing of a series T(I) until convergence is obtained:

```
SUM := 0.0;
while ABS(T(I)) > F'EPSILON * SUM loop
   SUM := SUM + T(I);
   I := I + I;
end loop;
```

As written it will stop summation appropriate to the declared properties of F. However, on a particular machine more accuracy might be obtained by writing F'BASE'EPSILON - going further than that would be pointless. Of course, a numerical analyst would sum such a series from the smallest term upwards, but the principle

remains the same.

Exercises

What is the relationship between F'DIGITS and F'BASE'DIGITS?

What is the relationship between F'LARGE and F'BASE'LARGE?

If F'DIGITS = G'DIGITS does F'BASE'DIGITS = G'BASE'DIGITS?

7. Fixed point data types

With fixed point data types, the user specifies the maximum acceptable absolute error bound. It is also necessary to specify the total range of values that must be covered, since the range and the error bound are required to determine the representation of values. A fixed point data type has the form:

```
type FX is delta D range L .. U;
```

where the D, L and U are static real expressions. All three values can be accessed as attributes of the fixed point type:

D = FX'DELTA, the absolute error bound,
L = FX'FIRST, smallest value of the type,
U = FX'LAST, the largest value of the type.

The type definition, together with a possible representation specification determines the set of model numbers of the type. The model numbers of the type are integer multiples of a value called the actual delta, which is an attribute of the type (=FX'ACTUAL_DELTA). This value is smaller than or equal to FX'DELTA so that values can be represented to within the accuracy specified. The range of integer values for the model numbers must be sufficient to be within FX'DELTA of both L and U. In an analogous way to the mantissa for floating point, the integer multiple (including the sign) is assumed to have a total range of -2**N+1.. 2**N-1 for some N. Hence to summarise, the model numbers are: (28)

```
sign * multiple * FX'ACTUAL_DELTA
```

where 0 <= multiple <= 2 ** N - 1 (for some N).

There are some essential differences here between fixed point and floating point. In floating point, some values such as 1.0 are always model numbers. This is not the case with fixed point. The value 1.0 could be less than FX'ACTUAL_DELTA and in consequence represented as 0.0. Conversely, 1.0 could exceed the range of values of the type and hence use of such a value could raise CONSTRAINT_ERROR. In general, unless a representation specification has been given which explicitly states the value of

ACTUAL DELTA, the model numbers are unknown. Hence one cannot assume that certain values, such as powers of two which are in range, will be represented exactly.

Let us now consider a practical example. The need is to process data which has an observational error of 0.01 and a range of 0.0 to 100.0. To ensure the ability to hold negative values, the type definition could be

```
type F is delta 0.01 range -100.0 .. 100.0;
```

A typical implementation could then choose a power of two for the ACTUAL_DELTA. This could be 1.0/128 or a smaller power, depending upon the word length of the machine. In this case, assume that F'ACTUAL_DELTA = 1.0/128. Then the model numbers are multiples of this value to a limit which must be at least within the range -100.0 + 0.01 .. 100.0 - 0.01. Since the multiples are a power of two, the model numbers are: (29)

```
M * 1.0/128 where -2**14 < M < 2**14.
```

Hence the largest model number is 128.0 - 1.0/128. This model number is outside the range of the type and in consequence cannot be assigned to values of type F.

Ordinarily, the ACTUAL_DELTA value is choosen by the compiler, the only constraint being that it must be less than or equal to the delta for the type. In a respresentation specification, the user may specify the ACTUAL_DELTA value. By such a specification, the representation of values can be constrained to conform to external requirements. For instance, if an analogue to digital converter from a camera places values in the memory of the computer, it is important that the Ada program should use the same representation. Consider the case when values 0 to 127 are input in binary, but these are regarded as fractions of unit intensity from 0.0 to 127.0/128. Then one might have:

```
type INTENSITY is delta 1.0/128 range 0.0 .. 127.0/128;
for INTENSITY'ACTUAL_DELTA use INTENSITY'DELTA;
for INTENSITY'SIZE use 7;
```

Note that -1.0/128 is a model number of the type but that it cannot be stored in values because of the range constraint and in consequence, the sign is not needed in the representation.

As a further example of a representation specification, consider handling a type analogous to DURATION in Ada ie, timing intervals. The obvious representation is in clock ticks so that the ACTUAL_DELTA value might be 1.0/60 seconds. The programmer would wish to work in seconds to avoid changing the program to work in Europe (with 50 cycle mains supply). Hence one might have:

```
type DURATION is delta 1.0/60 range -24.0 .. 24.0;
for DURATION'ACTUAL_DELTA use DURATION'DELTA;
```

The special attributes of a fixed point types are as follows:

F'DELTA: A real literal value equal to the value of the expression after the 'delta'. In this case the value is 0.01. (30)

F'ACTUAL_DELTA: The real literal value used by the implementation as the constant for the multiples which give the model numbers. In this case the value is 1.0/128 = 0.0078125.

F'BITS: This is the number of bits needed to represent the unsigned model numbers. In this case, 7 bits is required before the point and 7 bits after making 14 in all, ie F'BITS=14. The value is that of the integer literals (see section 9).

F'LARGE: The largest model number of the type F. In this case the value is 128.0 - 1.0/128 = 127.992175. The value has the same type as that of real literals. In general, one has F'LARGE = (2**F'BITS - 1) * F'ACTUAL_DELTA.

Subtypes of fixed point types can be declared explicitly or implicitly by giving an accuracy constraint on the declaration of a variable. In an exactly analogous way to floating point, there is no run-time check for an accuracy constraint for fixed point. Consider:

```
subtype SF is F delta 0.02;
```

Note that the range constraint is not needed since the range of values is determined from the type definition (-100.0 .. 100.0 in this case). This subtype definition means that the set of model numbers is correspondingly reduced by the value SF'ACTUAL_DELTA being a binary power multiple of F'ACTUAL_DELTA. In this case, with F'ACTUAL_DELTA = 1.0/128, SF'ACTUAL_DELTA could be 1.0/64. The implementation need not reduce the model numbers for a subtype. For this reason, it is not permitted to set the ACTUAL_DELTA for a subtype in a representation specification. Hence the only action required by the compiler for the above subtype declaration is to check that the expression after delta has a value greater than or equal to F'DELTA. If an implementation does reduce the model numbers for a subtype, then the values SF'BITS and SF'LARGE reflect the value of SF'ACTUAL_DELTA.

Exercises

Given: type FX is delta D range L .. U;

(a) Are D, L and U model numbers? (31)

(b) Can the range constraint be omitted?

(c) If L < 1.0 < U, is 1.0 a model number?

What is wrong with the following?

(d) type FD is delta 0.01 range 0.0 .. SQRT(2.0);

(e) type FE is delta 0.01 range 0 .. 10;

(f) type FF is delta 10.0 range 0.0 .. 100*FF'DELTA;

8. The predefined fixed point operations

With floating point, the specification of each operation was easy since with the exception of **, the type of the operands and the result was the same. It is easy to see that in general this is not possible with fixed point. The rescaling of intermediate results is done by explicit type conversion. This rescaling is only essential on multiplication and division since the magnitude of values only changes significantly with these operations.

The operations which do not involve rescaling are tabulated below. Here X is of any fixed point type and I of any integer type, the result always being the same as X. (32)

Example	meaning
+ X	no operation
- X	change sign
X + X	addition
X - X	subtraction
I * X	equivalent to repeated addition
X * I	equivalent to repeated addition
ABS(X)	absolute value
X / I	division without rescaling

Now consider some examples of the calculation of error bounds for computation using the same example type F as above:

```
type F is delta 0.01 range -100.0 .. 100.0;
```

Given

```
ONE : F := 1.0;
```

then it is not safe to assume that ONE is represented exactly

unless it is known that F'ACTUAL_DELTA is a submultiple of 1.0. An implementation is free to truncate or round a constant on conversion to F, and in consequence all one can say is that ABS(ONE-1.0) <= F'DELTA. To simplfy the remaining discussion, assume that F'ACTUAL_DELTA is 1.0/128, Then, of course, 1.0 is a model number and therefore is stored exactly. Now consider

```
TENTH : F := 0.1;
```

Here the value is bounded by the model interval 12.0/128 .. 13.0/128, and could be either of the two extremes or a value in between. Now consider the expression 10*TENTH. This is equivalent to repeated addition and in consequence can yield any value in the interval 120.0/128 .. 130.0/128. Of course, on a binary machine, the expression will never yield 1.0 exactly since 0.1 has a recurring binary representation.

The fixed point operations follow the same logic as for floating point as far as the definition of error bounds is concerned. In consequence, all of the operations above except, possibly, division by an integer, will yield exact results for multiples of F'ACTUAL_DELTA assumming the result is in range. Since in some cases, the implementation will not have values between model numbers, this implies that all these operations are exact. In fact, values between model numbers can only arise from division by an integer, from constant, and type conversions from other types. The nature of the inexact operation can be illustrated from

```
X: F := 10.1;
Y: F := X/2;
```

The X value is bounded by the interval 1292.0/128 .. 1293.0/128. The Y value is then bounded by the interval 646.0/128 .. 647.0/128. Hence multiplication of Y by 2 will yield a larger bounding interval than that of X. If, of course, one knew that X was equal to an even multiple of F'ACTUAL_DELTA, then Y := X/2; and X := Y*2; would leave X unchanged. (*1) The model numbers for an integer type in fixed point operations are just the integers themselves. (33)

The rescaling operations are general fixed point multiplication and division. The operands are of any, possibly distinct, fixed point types. Consider the types:

```
type F is delta 0.01 range -100.0 .. 100.0;
type G is delta 1.0 range -10_000.0 .. 10_000.0;
```

Given F1,F2:F, consider the product F1 * F2. This product is very likely to overflow the range of F and hence in general it cannot be considered to be of type F. Using the intuitive concept of formats, it is quite clear that a product has a different format.

(*1) If the statements Y:=X/2; X:=Y*2; appeared in a program, compiler optimization could give the exact result in all cases (since CONSTRAINT_ERROR cannot arise, optimization is safe).

On the other hand, given G1:G, it is clear that one should be able to assign the product to G1 without overflow. In Ada, the product is regarded as Universal Fixed - a hypothetical type of arbitrarily high accuracy. This type does not have an Ada name so that no variables can be declared of this type. All that can be done with such a product is to convert it into another type. In this case, one can write: (34)

```
G1 := G(F1 * F2);
```

To calculate error bounds, the operation is regarded as a whole: ie calculating the product and the conversion. Assuming that F'ACTUAL_DELTA = 1.0/128 and G'ACTUAL_DELTA = 1.0, then one has:

F1	1.0	10.0	10.1
F2	2.0	0.1	10.1
bounds on F1			
	1.0..1.0	10.0..10.0	1292.0/128..1293.0/128
bounds on F2			
	2.0..2.0	12.0/128..13.0/128	1292.0/128..1293.0/128
bounds on F1*F2			
	2.0..2.0	120.0/128..130.0/128	101.88 .. 102.04
bounds on G1			
	2.0..2.0	0.0 .. 2.0	101.0 .. 103.0

In this case, the error bounds are about 2 units in the resulting type. Clearly, if the operands have high accuracy as well as the resulting type for the product, then no accuracy need be lost.

Fixed point division works in the same way with the requirement to convert to result of the operation. Naturally, if the right hand operand has a small value, then substantial inaccuracies can occur.

Exercises

(a) Given a fixed point type with a range of positive values only, can the function ABS have any use?

(b) Should an implementation limit the smallness of the delta value?

(c) What purpose does the 'DELTA value serve if errors are bounded by multiples of 'ACTUAL_DELTA?

(d) Given type FD is delta 0.01 range -1.0 .. 1.0; and assuming FD'ACTUAL_DELTA = 1.0/128, and X = 0.1, calculate the error bounds

on:

```
Y := 0.6 + FD(0.2*X) + FD(0.1*FD(X*X));
```

and on

```
Y := FD((FD(0.1*X) + 0.2)*X) + 0.6;
```

9. Literal expressions

In the preceeding sections one problem has been avoided, namely, the type of a numeric literal. It has been noted tnat literals are implicitly converted to the type required by the context. This implicit conversion can lose accuracy and can also raise the exception CONSTRAINT_ERROR if the value exceeds the implemented range of the type. (35)

Integer literals are regarded as being of type Universal Integer and real literals of type Universal Real. The names of these types are not available as Ada names and in consequence, cannot be used to declare variables etc. However, it often happens, especially with fixed point working, that there are relationships between literal values which cannot be easily expressed be means of typed expressions. Ada therfore allows for the evaluation (by the compiler) of literal expressions. Hence, wherever Ada permits an integer expression 1+1 (say), can be written. Each literal 1 is of type Universal Integer and the "+" is evaluated by the compiler independently of the context. Consider the following:

```
type INT is range -10 .. 10;
TEN: constant INT := 10;
THOUSAND: constant := 1000;
I : INT;

I := TEN;   -- OK
I := 1000 * TEN;    -- (1)
I := THOUSAND - THOUSAND;  -- (2) OK I := 0;
I := THOUSAND - 1000;   -- (3)
```

In case (1), the actions are as follows: 1000 is implicitly converted to INT, the INT "*" operation is applied, the result is checked for range, and lastly the assignment is performed. In this case, the first step fails as 1000 is not within the range of the type, and hence CONSTRAINT_ERROR is raised.

In cases (2) and (3), the result is to assign zero to I since the subtraction is that of Universal Integer which is performed by the compiler (within an unbounded range).

Similar remarks apply to real literal expressions:

```
type F is digits 5;
RATIO: constant F := 3.14;
```

```
PI: constant := 3.14159_26535;
F1 : F;

F1 := 2.0 * PI;  -- Universal Real multiplication

F1 := 2.0 * RATIO; -- Multiplication of type F

F1 := 1.0E200 - 10.0 * 1.0E199;
                  -- Literal expression value 0.0
                  --  no overflow possible
```

Of course, with both Universal Integer and Universal Real, a compiler will have some limitation in the size of values and accuracy respectively. These limits should not be of any practical significance and in consequence will be larger than any implemented type available on typical target machines.

The type Universal Real is not strictly a floating point or fixed point type but has the functionality of both. In consequence, all the following operations are legal as illustrated by the number declarations:

```
ADD: constant := 1.0 + 3.0;
SUB: constant := 6.0 - 8.0;
PLUS: constant := +12.0;
NEG: constant := - 8.4;
MULT1: constant := 2 * 3.0;
MULT2: constant := 4.0 * 10;
MULT3: constant := 5.0 * 10.1;
DIV1: constant := 10.0/2;
DIV2: constant := 10.0/2.0;
EXP: constant := 3.0 ** 2;
ABS1: constant := ABS(6.0);
```

(36)

The relational operators are also available for Universal Real giving the BOOLEAN result as usual. The semantics of these operations is the same as that for the typed operations except that the model intervals are smaller than any implemented real type. Intuitively, one can envisage a compiler storing values in a floating point type of high accuracy.

Exercises

Are the following literal expressions?

(a) F'DIGITS + 10
(b) FX'DELTA / 10.0
(c) FX'LAST - 0.1
(d) 1/FX'ACTUAL_DELTA

(37)

10. A floating point example, Blue's algorithm

Blue's algorithm [5] is for the calculation of the Euclidean norm of a vector (ie square root of the sum of the squares of the elements). It is a natural extension of the calculation of SQRT(A**2+B**2) which was used above. The algorithm is very carefully written to avoid overflow and underflow and also to guarantee the precision of the result. Hence it is a good example of a high-quality algorithm (orginally written in FORTRAN). The paper itself should be studied for the numerical analysis involved. (38)

The paper presents the algorithm in two forms: a mathematical formulation using Greek letters and conventional notation; and a formulation in RATFOR, a FORTRAN preprocessor. The major differences between the RATFOR version and that for Ada below are that the Ada version works for any implemented precision and does not depend on an additional subroutine to set critial constants. In Ada, these constants are model numbers and hence the definition of the language guarantees that the values are set correctly from the predefined attributes.

The identifiers used are those of the RATFOR implementation, but in upper case. In practice, longer identifiers that show the relationship to the mathematical formulation of the algorithm would be preferable.

The function itself, called NORM must assume an appropriate context for the types of its parameters and for the SQRT routine. This context is:

```
type REAL is digits D;
type VECTOR is array (INTEGER) of REAL;
function SQRT(X: REAL) return REAL;
```

The body of the function can now be given. The main logic is to go once through the vector accumulating three sums according to the magnitude of each element. The three sums are then scaled as appropriate to give the final answer.

```
function NORM(X: VECTOR) return REAL is

   -- calculate constants which depend upon REAL.
   -- Floor and ceiling functions of the paper avoided
   --    by using the truncation of integer division.
   EB1: constant := (REAL'EMAX + 1)/2; --  -(exponent of B1)
   B1 : constant REAL := 2.0 **(-EB1); -- model number of REAL

   EB2: constant := (REAL'EMAX - REAL'MANTISSA + 1)/2;
   B2 : constant REAL := 2.0 ** EB2;

   ES1M: constant := REAL'EMAX/2 + 1;
```

(39)

```
   S1M : constant REAL := 2.0 ** ES1M;

   ES2M: constant := (REAL'EMAX + REAL'MANTISSA + 1)/2;
   S2M : constant REAL := 2.0 ** (-ES2M);
   OVERFL: constant REAL := REAL'LARGE * S2M;
               -- this value can be calculated by the compiler

   RELERR: constant REAL := SQRT(REAL(REAL'EPSILON));
           -- Conversion necessary as 'EPSILON is a literal.
           -- Note that this value must be calculated dynamically.

   ABIG, AMED, ASML: REAL := 0.0; -- the three accumulators
   AX: REAL;
   N: constant INTEGER := X'LENGTH;
begin
-- size of array = 0 is not special case in Ada (unlike FORTRAN)
if N > 2**REAL'MANTISSA then                                    (40)
   raise CONSTRAINT_ERROR;  -- not clear how to handle this case
end if;
for J in X'FIRST .. X'LAST loop
   AX := ABS(X(J));
   if AX > B2 then
      ABIG := ABIG + (AX * S2M)**2;
   elsif AX < B1 then
      ASML := ASML + (AX * S1M)**2;
   else
      AMED := AMED + AX**2;
   end if;
end loop;
if ABIG > 0.0 then
   ABIG := SQRT(ABIG);
   if ABIG > OVERFL then
      return REAL'LARGE; -- can't raise NUMERIC_ERROR as well as
                         -- returning a result
   end if;
   if AMED > 0.0 then
      ABIG := ABIG / S2M;
      AMED := SQRT(AMED);
   else
      return ABIG/S2M;
   end if;
elsif ASML > 0.0 then
   if AMED > 0.0 then
      ABIG := SQRT(AMED);
      AMED := SQRT(ASML)/S1M;
   else
      return SQRT(ASML)/S1M;
   end if;
else
   return SQRT(AMED);     -- the standard path
end if;
if ABIG > AMED then
   ASML := AMED;
```

```
else
   ASML := ABIG;
   ABIG := AMED;
end if;
if ASML <= ABIG * RELERR then
   return ABIG;
else
   return ABIG * SQRT( 1.0 + (ASML/ABIG)**2 );
end if;
end NORM; --  (*1)
```

(41)

The algorithm is not completely satisfactory in the sense that although it will work for any real type, the algorithm uses only the Ada properties of the type. By replacing each occurance of 'REAL by REAL'BASE, the algorithm would use the properties of the implemented type. With such a replacement, there would only be one effective version of the function for each of the hardware types. Further 'improvements' can be made by exploiting specific machine-dependent properties of the type (see section 13).

(42)

11. A fixed point example

A common requirement in fixed point is to mimic floating point to conserve either time or space. The evaluation of a polynomial is sometimes used to approximate a function. Such polynomials are typically truncated power series which rely upon the decreasing contributions from the higher order terms. Given:

(43)

```
Y := A + B*X + C*X**2 + D*X**3;
```

the most effective evaluation method is nested multiplication, ie

```
Y := ((D*X + C)*X + B)*X + A;
```

As X is small, with floating point, the normalization on the addition of A is the only source of rounding error. With fixed point using a pure fraction as the data type, each partial product can be calculated with minimal errors so that the resulting error is again minimised. Note that performing the calculation in polynomial fashion both involves more operations and is, in general, less accurate.

To illustrate the use of fixed point for approximation, the calculation of the sine and cosine functions is given from Cody and Waite [6]. It is assumed that floating point is expensive on the target machine and therefore the major computation uses fixed point. The algorithm illustrates a number of other features including type conversion, literal expressions, integer type definitions and conditional compilation. The algorithms given in [6] include a number of options which would ordinarily be chosen by the implementor. Some of the choices in this case are inserted into the algorithm so that the compiler selects the necessary code.

(*1) Not yet tested with an Ada compiler

In order to correspond to conventional usage, the routines SIN and COS are coded for the type FLOAT. No assumptions are made about the type except that integer and fixed point types are available with sizes about the same as the mantissa length of the type FLOAT.

The argument reduction is a difficult aspect of sine and cosine. The constants C1 and C2 (whose sum is PI) are used for this in the way recommended for a machine without guard digits on floating point. The code illustrates that compiler "optimization" of floating point can be very unsafe.

```
package SIN_COS is
    function SIN( X: FLOAT) return FLOAT;
    function COS( X: FLOAT) return FLOAT;
end SIN_COS;

package body SIN_COS is
    PI: constant := 3.14159_26535_89793_23846;                    (46)
    PI_DIV_2: constant := PI/2;
    ONE_DIV_PI: constant := 1.0/PI;
    SGN_POS: BOOLEAN;
    Y: FLOAT;

procedure COMMON_PART( X: FLOAT );

function SIN( X: FLOAT) return FLOAT is
    begin
    if X < 0.0 then
        SGN_POS := FALSE;
        Y := - X;
    else
        SGN_POS := TRUE;
        Y := X;
    end if;
    COMMON_PART(X);
    return Y;
    end SIN;

function COS( X: FLOAT ) return FLOAT is
    begin
    SGN_POS := TRUE;
    Y := ABS(X) + PI_DIV_2;
    COMMON_PART(X);
    return Y;
    end COS;

procedure COMMON_PART( X: FLOAT ) is
    B: constant := FLOAT'MANTISSA;                                (46)

    type INT is range 0 .. 4 * 2**(B/2);
    YMAX: constant INT := INT(PI*2**(B/2)+0.5);
```

```
N: INT;

X1, X2, XN, F: FLOAT;
C1: constant FLOAT := 8#3.1104#;
C2: constant FLOAT := -8.9089_10206_76153_73566_17E-6;
EPS: constant := 2.0 ** (B/2);

D: constant := 2.0 ** (-B);
type FR is delta D range -1.0 + D .. 1.0 - D;
G: FR;

begin
if Y >= FLOAT(YMAX) then
    raise CONSTRAINT_ERROR;
else
    N := INT( Y * ONE_DIV_PI );
    XN := FLOAT(N);
    if N mod 2 = 1 then
        SGN_POS := not SGN_POS;
    end if;
    if ABS(X) /= Y then
        XN := XN - 0.5;  -- COS wanted
    end if;
    X1 := FLOAT(INT(ABS(X)));
    X2 := ABS(X) - X1;
    F := ((X1 - XN*C1) + X2) - XN*C2;                    (47)
    if ABS(F) < EPS then
        Y := F;
    else
        G := FR(F/2.0);
        G := FR(G * G);
        if B <= 24 then                                  (48)
            Y := FLOAT(
                   FR((
                     FR((
                       FR((
                         FR(0.00066_60872 * G)
                       - 0.01267_67480) * G)             (44)
                     + 0.13332_84022) * G)
                   - 0.66666_62674) * G)
                   );
        elsif B <= 32 then
            Y := FLOAT(
                   FR((
                     FR((
                       FR((
                         FR((
                           FR(-0.00002_44411_867 * G)
                         + 0.00070_46136_593) * G)
                       - 0.01269_81330_068) * G)
                     + 0.13333_32915_289) * G)
                   - 0.66666_66643_530) * G)
                   );
```

```
            elsif B <= 50 then
                Y := FLOAT(
                        FR((
                          FR((
                            FR((
                              FR((
                                FR((
                                  FR((
                                    FR(-0.00120_76093_891E-5 * G)
                                  + 0.06573_19716_142E-5) * G)
                                - 0.00002_56531_15784_674) * G)
                              + 0.00070_54673_00385_092) * G)
                            - 0.01269_84126_86862_404) * G)
                          + 0.13333_33333_32414_742) * G)
                        - 0.66666_66666_66638_613) * G)
                        );
            elsif B <= 60 then
                Y := FLOAT(
                        FR((
                          FR((
                            FR((
                              FR((
                                FR((
                                  FR((
                                    FR(0.00001_78289_31802E-5 * G)
                                  - 0.00125_22156_53481E-5) * G)
                                + 0.06577_74038_64562E-5) * G)
                              - 0.00002_56533_57361_43317) * G)
                            + 0.00070_54673_71779_91056) * G)
                          - 0.01269_84126_98369_17789) * G)
                        + 0.13333_33333_33330_64050) * G)
                      - 0.66666_66666_66666_60209) * G)
                      );
            else
                raise CONSTRAINT_ERROR;
            end if;
            Y := F + F*R;
        end if;
        if not SGN_POS then
            Y := - Y;
        end if;

    end COMMON_PART;

end SIN_COS;  --  (*1)
```

(*1) Not yet tested with an Ada compiler

12. The complex data type - an example of generics

An essential difficulty with both fixed point and floating point subroutines is that they can only be used with one type - and in consequence one accuracy. Usually, the text of a subroutine will be more general than this although when it is compiled, it will be for a specific accuracy. The full generality of the source text can be exploited by means of generics. The numeric types are made generic parameters so that specific instantiations give any specific accuracy (supported by the implementation).

As an example of generics, the package for providing complex data types is used. This is very similar to the rational number package given in the language reference manual (which does not use generics).

```
generic
    type REAL is digits <>;  -- matches any floating point type
package COMPLEX_OPS is
    type COMPLEX is record
                    RE, IM: REAL;
                    end record;
    function "-" ( X: COMPLEX) return COMPLEX;
    function ABS( X: COMPLEX) return REAL;
    function "+" ( X, Y: COMPLEX ) return COMPLEX;
    function "-" ( X, Y: COMPLEX ) return COMPLEX;
    function "*" ( X, Y: COMPLEX ) return COMPLEX;
    function "/" ( X, Y: COMPLEX ) return COMPLEX;
end COMPLEX_OPS;
```

The package body does not repeat the generic parameters, and could be:

```
with MATH_LIB;
package body COMPLEX_OPS is

    function "-" ( X: COMPLEX) return COMPLEX is
        begin
        return ( - X.RE, - X.IM );
        end "-";

    function ABS( X: COMPLEX) return REAL is
        A, B: REAL;
        begin
        if ABS(X.RE) > ABS(X.IM) then
            A := ABS(X.RE);
            B := ABS(X.IM);
        else
            A := ABS(X.IM);
            B := ABS(X.RE);
        end if;
        if A > 0.0 then
```

```
            return A * MATH_LIB.SQRT(1.0 + (B/A)**2);
        else
            return 0.0;
        end if;
        end ABS;
    function "+" ( X, Y: COMPLEX ) return COMPLEX is
        begin
        return ( X.RE + Y.RE, X.IM + Y.IM );
        end "+";

    function "-" ( X, Y: COMPLEX ) return COMPLEX is
        begin
        return ( X.RE - Y.RE, X.IM - Y.IM );
        end "-";

    function "*" ( X, Y: COMPLEX ) return COMPLEX is
        begin
        return ( X.RE*Y.RE - X.IM*Y.IM,
                 X.IM*Y.RE + X.RE*Y.IM );
        end "*";

    function "/" ( X, Y: COMPLEX ) return COMPLEX is
        A: REAL := Y.RE**2 + Y.IM**2;
        begin
        return ( (X.RE*Y.RE + X.IM*Y.IM) / A,
                 (X.IM*Y.RE - X.RE*Y.IM) / A );
        end "/";

end COMPLEX_OPS;  --  (*1)
```

(50)

13. Portability Issues

The Ada language does not aim at complete portability. To do so would mean that it would be impossible to write machine-specific code such as that illustrated in section 11. Also, the differences in actual hardware does not make portability achievable at acceptable costs. Ultimately, the floating point addition of a machine is defined by the microcode, which cannot even be characterized by a few simple parameters. Hence programmers need to be aware of potential portability problems so that code is not needlessly machine-specific.

(51)

Integer Types.
New types should be introduced for reasons of abstraction and modularization. It is particularly important to introduce new integer types in handling large ranges (> 16 bits) since these may not be supported, or will have some significant penalty. If a large integer type is only used in one small routine, then recoding is much simpler than if INTEGER is used thoughout (and

---

(*1) Not yet tested with an Ada compiler

only during execution it is found that one routine assumes INTEGER'LAST > 2**16).

A trap for the unwary is that intermediate results in an expression of type T can exceed the range T'FIRST..T'LAST. In consequence, portability is not assured since on other hardware T may correspond exactly to the range of a hardware type. Diagnostic compilers can trap this condition and cause NUMERIC_ERROR to be raised at run-time. Of course, the values for which NUMERIC_ERROR is raised is again dependent upon the hardware. On some machines, calculations are done to 32 bits but stored values are 16 bits giving the overlength accumulator problems analogous to floating point.

Floating Point Types.
Similar difficulties arise with the handling of NUMERIC_ERROR for floating point. The actual range of the implemented type is likely to exceed the range -F'LARGE .. F'LARGE due to having a larger exponent than that guaranteed by Ada. A machine may have 'infinite' values such as those of the IEEE standard [7], in which case substantial care is necessary to ensure such facilities are avoided or used in a portable fashion. For this reason, the values F'FIRST and F'LAST should be avoided. It must be remembered that the NUMERIC_ERROR exception might never be raised.

Explicit use of the type FLOAT should be restricted to small sections of code. It would be reasonable to assume that FLOAT has 5 digits of precision. Of course, some machines may not have any floating point. Apart from the use as a tool for abstraction, new types should be introduced when different accuracies are needed. The use of accuracy constraints in subtypes should only be regarded as a comment, rather than attempting to rely upon sophisticated optimization.

A set of machine specific attributes for floating point is available which, if used, is unlikely to give portable code. For this reason, these attributes have names beginning with MACHINE_. They are as follows:

F'MACHINE_RADIX. This is the radix used to represent machine values. To support the Ada model of floating point properly, it must be a power of 2. It is 16 for the IBM 360/370 and 2 for the IEEE standard.

F'MACHINE_MANTISSA. This is the mantissa length in radix units. It is at least conceivable that there is not a whole number of radix places in the mantissa, although the value is defined to be an integer.

F'MACHINE_EMAX. The maximum exponent value in radix units. (So F'MACHINE_RADIX ** F'MACHINE_EMAX is

approximately the largest machine number.)

F'MACHINE_EMIN. The minimum exponent value in radix units.

F'MACHINE_ROUNDS. A BOOLEAN value which is true only if all floating point operations perform true rounding, such as that of the IEEE standard.

F'MACHINE_OVERFLOWS. A BOOLEAN value which is true only if the NUMERIC_ERROR exception is raised whenever the result of an operation cannot be represented with the usual precision due to exceeding the range of machine values.

With care, the MACHINE_RADIX value can be used to overcome the problem of 'wobbling precision' and the attribute MACHINE_OVERFLOWS can be used to provide an alternative coding which relies upon the NUMERIC_ERROR exception. Note that one cannot easily determine the largest and smallest positive machine values due to the differences between 1's and 2's complement arithmetic, underflow etc.

Fixed Point Types.
A similar remark applies to fixed point about not relying upon values outside the range -LARGE..LARGE. A program can also depend upon the arithmetic of the machine. With a pure fraction, on a 2's complement machine, -1.0 is a machine number but 1.0 is not, whereas neither are machine values on a 1's complement machine. Although it is highly likely that the default value for ACTUAL_DELTA will be a power of two, the program should not rely upon this.

In the same way that one should not rely upon the existance of floating point with a large number of digits, so with fixed point one should not expect very high precision (exceeding 32 bits, say).

Fixed point types have the attribute FX'MACHINE_ROUNDS which is true only if all the operations perform true rounding.

References

[1] W S Brown. "A realistic model of floating point computation" Mathematical Software III ed. J Rice pp 343-360 Academic Press New York 1977.

[2] W S Brown. "A simple but realistic model of floating point computation". Computer Science Technical Report No. 83, May 1980. Bell Laboratories, Murray Hill, NJ 07974.

[3] W S Brown and S I Feldman. "Environmental parameters and basic functions for floating point computation". Computer Science Technical Report No. 12, March 1980, Bell Laboratories, Murray Hill, NJ 07974

[4] J D Ichbiah et al. "Reference Manual for the Ada programming language". Department of Defense, Washington, July 1980.

[5] J L Blue. "A portable FORTRAN program to find the Euclidean norm of a vector." ACM TOMS Vol4, No1 pp15-23 March 1978.

[6] W J Cody and W Waite. "Software manual for the elementary functions." Prentice-Hall. Englewood Cliffs, NJ 07632. 1980.

[7] "A Proposed Standard for Binary Floating Point Arithmetic" SIGNUM Newsletter, October 1979. (Now to be adopted as an IEEE Standard.)

Answers to exercises

Page 5 (section 2)

16#FF# = 15 * 16 + 15 = 240 + 15 = 255
4#1.01#E2 = 4#101.0# = 4.0 ** 2 + 1.0 = 17.0
3#0.1# = 3.0 ** (-1) = 0.33333 (no exact decimal equivalent)
8#0.1# = 8.0 ** (-1) = 0.125
16#0.8# = 8* 16.0 ** (-1) = 0.5
16#0.999999...# = 9.0/(16-1) = 3.0/5 = 0.6
3._14 — no underscore after decimal point
4#_0.1#2 — no underscore after sharp
16#FF#E-1 — not integer valued and no decimal point
8#0.9# — 9 not a radix character.

Page 7 (section 3)

(a) Next model number above 1.0 is 16#0.8001# * 2
=16#1.0002#

(b) Next model number below 1.0 is 16#0.7FFF8#
= 16#0.FFFF#

(c) The ratio is 2.0 which is the radix.

The rational numbers which cannot be represented exactly are those with recurring binary representation.

Page 13 (section 4)

Not necessarily. If F'DIGITS = 14 then F'MANTISSA = 47 but G'DIGITS = 7 gives G'MANTISSA = 24.

X = F'LARGE since 1.0/F'LARGE > F'SMALL and hence does not underflow. Of course, the actual machine may permit larger and smaller values without over/underflow.

Page 16 (section 5)

(A+B)+C=A+(B+C) is not necessarily true since floating point addition is not associative. For values A, B and C which are model numbers such that the true sum (and partial sums) are model numbers, the result will be true.

A+B=B+A is usually true but is not necessarily so. A+B could be calculated in the accumulator of the machine and then stored while B+A is evaluated in the accumulator. The comparison may then fail if the accumulator gives more precision than that of stored values.

```
A + 12 -- 12 is not real, must write A + 12.0
24 * B -- 24 is not real, must write 24.0 * B
C ** 2.0 -- exponent must be integer, hence should be C**2
```

Page 18 (section 6)

F'BASE'DIGITS >= F'DIGITS
F'BASE'LARGE >= F'LARGE

Not necessarily, since if the short form is used (without new), the compiler is free to choose the hardware type which need not be the same.

Page 21 (section 7)

(a) Not necessarliy, but there must be model numbers close to L and U.
(b) No, it is required to determine the representation.
(c) Not necessarily, if the actual_delta is not a submultiple of 1.0, then 1.0 will not be a model number. The actual_delta value could exceed 1.0.
(d) The range must be static, hence the call of SQRT is not permitted.
(e) The range is a real range and hence should read "0.0 .. 10.0".
(f) The attribute 'DELTA is not defined until the end of the type definition and hence cannot be used within the definition.

Page 23 (section 8)

(a) Yes, values of the type always include negative values since model numbers can be negative. These negative values may cause CONSTRAINT_ERROR on assignment. If the sign of a small value is in doubt, ABS can be used before assignment.

(b) There should be no practical lower limit. An implementation is likely to limit the number of model numbers for a type so that values can be held in one or two words. Hence if the delta value is very small, the L and U values should be also.
(c) Since 'DELTA >= 'ACTUAL_DELTA, the bounds can be expressed in terms of DELTA (ie the type definition).
(d) In both cases, 0.6 is of type FD and bounded by 76.0/128 .. 77.0/128. The constants 0.1 and 0.2 are Universal Real and in the first case are held to the relative accuracy of FD ie, 7 bits. This implies that 0.1 is bounded by an interval of width 1.0/(8*128) and 0.2 by an interval twice that width.

```
CASE 1
0.6      bounded by      76.0/128 .. 77.0/128
FD(0.2*X)   bounded by     2.0/128 .. 3.0/128
P=FD(X*X)   bounded by     1.0/128 .. 2.0/128
FD(0.1*P)   bounded by     0.0/128 .. 1.0/128
Y           bounded by   (76.0+2.0+0.0)/128 .. (77.0+3.0+1.0)/128
                    =     78.0/128 .. 81.0/128

CASE 2
Q=FD(0.1*X)    bounded by 1.0/128 .. 2.0/128
R=Q + 0.2      bounded by 26.0/128 .. 28.0/128
T=FD(R*X)      bounded by 2.0/128 .. 3.0/128
Y=T+0.6        bounded by 78.0/128 .. 80.0/128
```

The effectiveness of nested multiplication increases with the number of terms, as can be seen from the relative error of the higher order terms.

Page 25 (section 9)

(a) Yes, F'DIGITS is Universal Integer.
(b) Yes, FX'DELTA is Universal Real.
(c) No, FX'LAST is of type FX.
(d) No, Universal_Integer/Universal_Real is not permitted.

Real Data Types in Ada

B. A. Wichmann.

National Physical Laboratory

UK

Real = Fixed or Floating Point

FORMATS

Fixed Point	±dd.d	±ddd.d
Floating Point	±d.ddE±dd	±d.dddE±dd

Data Type defines FORMAT

Fixed point ± d.dd

Examples: half + 0.50 third + 0.33 π + 3.14

stored exactly

error < 0.01

Floating point ± d.dd E ±d

Examples: hundred + 1.00E+2 π + 3.14E+0 third + 3.33E-1

stored exactly

error < 1 part in 1000

Fixed point – absolute error bound

Floating point – relative error bound

Some values stored exactly

Data Types to distinguish logically separate data.

Hence even if FEET and DEGREES both require format ±ddd.d, distinct types should be used.

Decimals formats are used for illustration only.
(Ada uses binary).

Notation for numeric literals in Ada

Integers - sequence of digits with underscore
(plus optional exponent)

Real literals - with decimal point

Six million

integer	6_000_000	6E6	6_000E+3
real	6_000_000.0	6.0E6	6_000.0E+3

Bases other than ten

2 # 101 # = $2^2 + 2^0 = 5$ integer

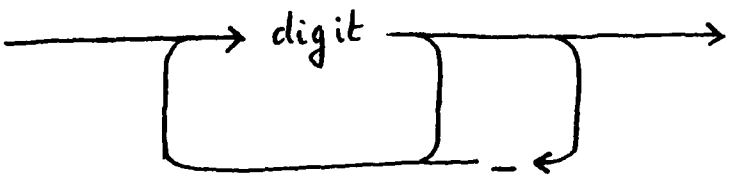
↑ base ↓ base digits

4 # 1.1 # = $4.0^{1} + 4.0^{-1} = 1.25$ real

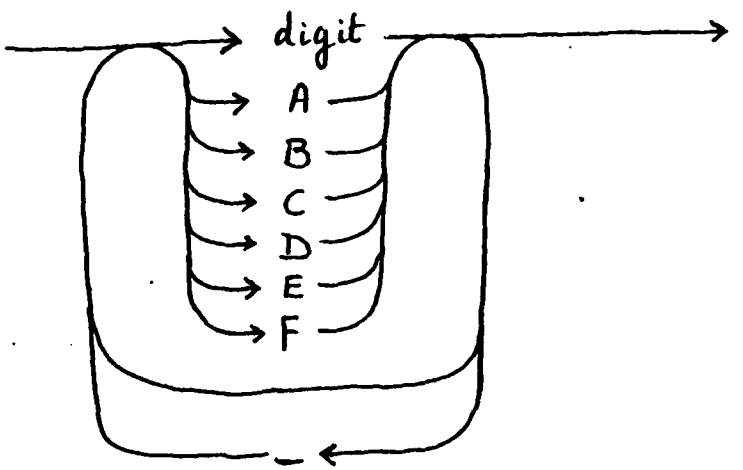
2 # 1.0 # E-20 = $2.0^{-20} \simeq 9.54E-7$ real

16 # F F # = $15 * 16^{1} + 15 * 16^{0} = 255$ integer

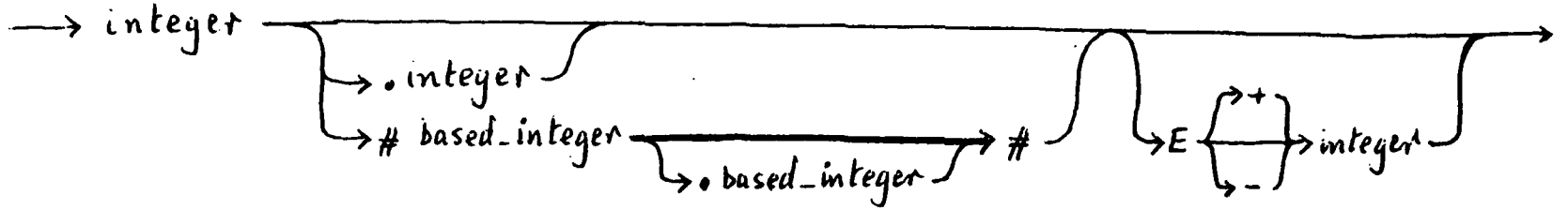
integer:



based_integer:



numeric_literal



number_declaration

→ identifier : constant := numeric_literal ; →

Example

```
PI : constant := 3.14159_26535_89793;
```

Exercises

What values are these in decimal?

16#FF# 4#1.01#E2 3#0.1# 8#0.1# 16#0.8#

What value is 16#0.999999# just a bit less than?

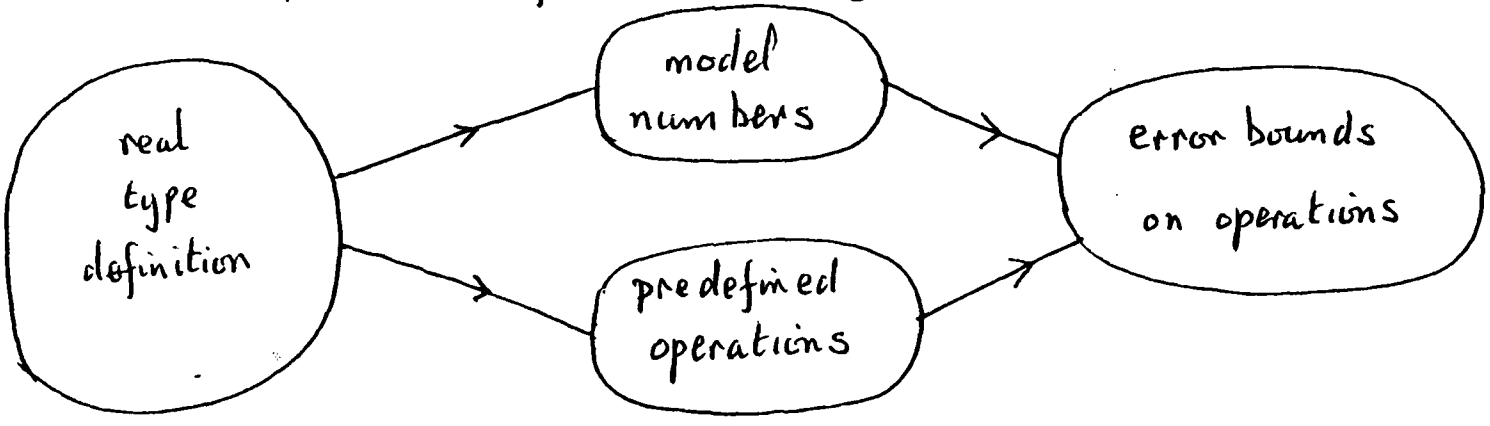
What is wrong with the following?

3._14 4#_0.1#2 16#FF#E-1 8#0.9#

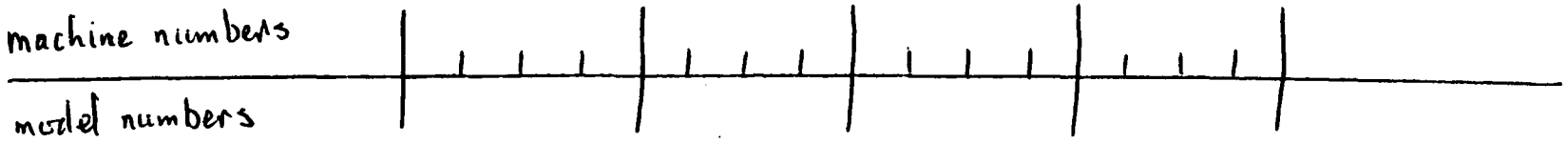
(Answers at back of notes)

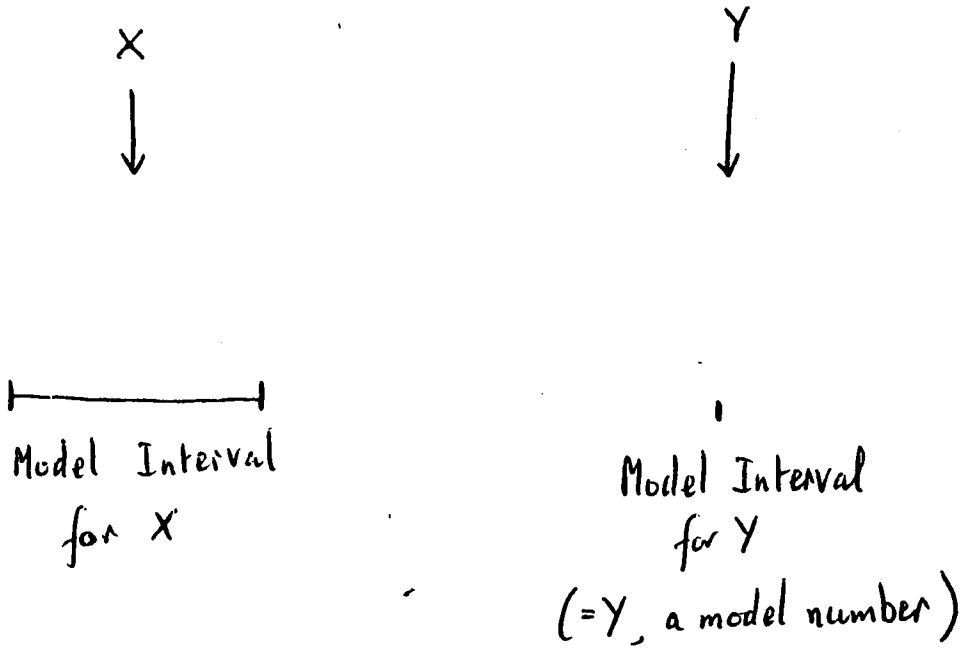
Ada properties of real types

- Permissive - different implementations allowed.
- Subset of values held exactly - model numbers
- Intervals bounded by model numbers bound errors in operations
- Model numbers defined in binary

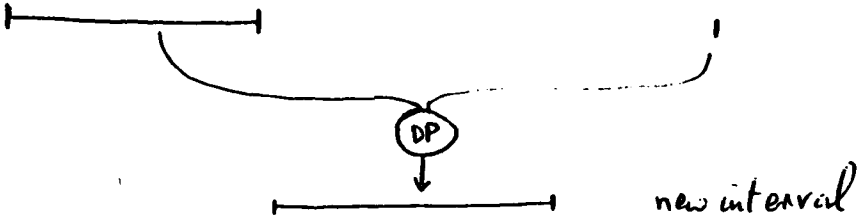


Accuracy of real operations



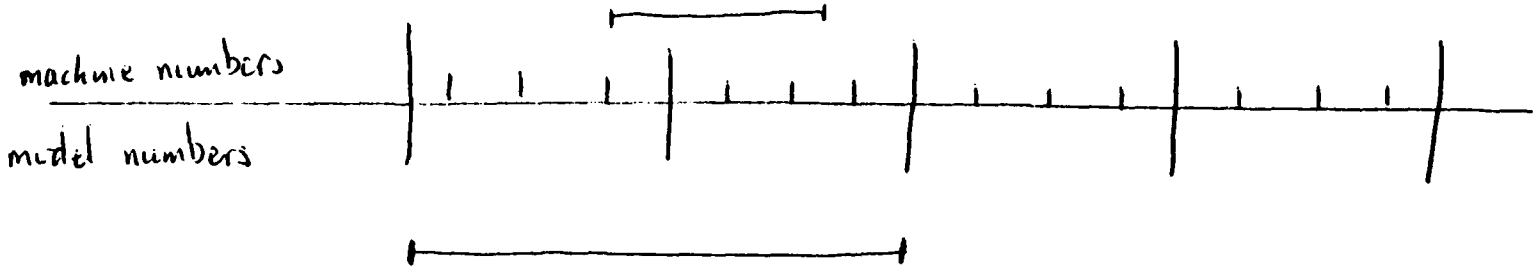


STEP 1, widen operands to model intervals



STEP 2, Perform operation mathematically to the intervals

(to be placed on top of 11, without 11A)



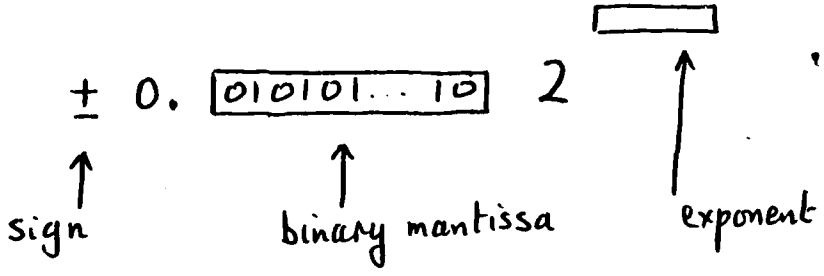
STEP 3, Widen resulting interval to a model interval.

This interval bounds the implemented operation.

Corollary 1: Floating point operations on small integers giving small integer results are exact.

Corollary 2: Classical error analysis can be applied.

Model Numbers for Floating Point



Type definition gives decimal digits of accuracy D.

implies D * log (10)/log (2) binary places

= B

Exponent range taken to be (at least)

- 4 * B .. 4 * B

(compromise based upon current hardware).

Example

type F is digits 5;

F'DIGITS = 5

requires 5 * 3.32 = 16.6 binary places

Hence B = 17

F'MANTISSA = 17

Exponent range is -68 .. 68

F'EMAX = 68

Smallest model number > 0.0 is

2.0 ** (-69) = F'SMALL

Largest model number is

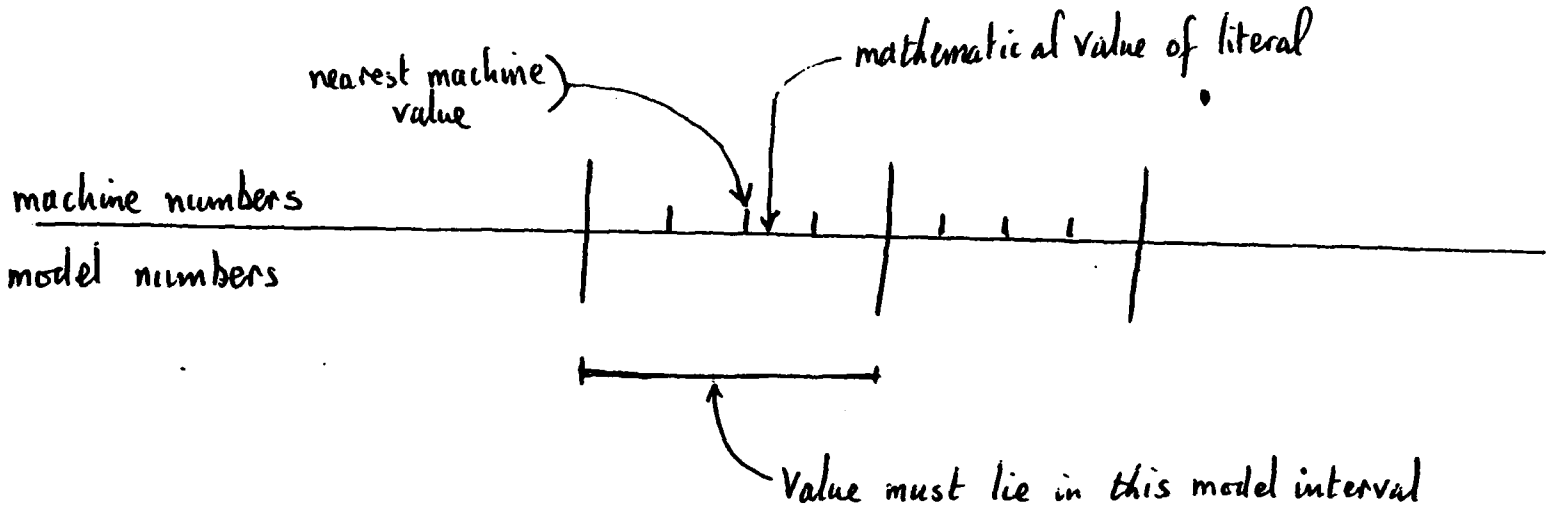
2#0.11111_11111_11111_11#E68

= 2.0 **68 - 2.0 **51 = F'LARGE

Gap between 1.0 and next model number above 1.0

= 2.0 ** (-16) = F'EPSILON

Conversion of literals



Corollary: Literals whose value is a model number are stored exactly.

Under flow $|X| < F'SMALL$

Model intervals are $-F'SMALL .. 0.0$

and $0.0 .. F'SMALL$

Value produced could be 0.0 or $\pm F'SMALL$

(large relative error)

Example:

```
X := SQRT(A **2 + B **2);
```

Small values for A and B will give X = 0.0 (or F'SMALL?)

Can be overcome by special coding if necessary

(See Notes)

OVERFLOW

Occurs in steps 1 and 3 in the definition of the bounding model interval. Values X, such that $|x| > F'LARGE$ have no bounding interval.

Several things can happen:

(a) Due to large exponent range, errors are bounded properly on actual hardware

(b) NUMERIC_ERROR exception is raised

(c) "infinite" values are produced by hardware. (These are not in any model interval)

Conversion of literal values will raise CONSTRAINT_ERROR if out of range of implemented type.

NUMERIC_ERROR and CONSTRAINT_ERROR

Exceptions are propogated through procedure calls

Hence specification of procedure should state

(a) Exceptions cannot be raised

(b) Local handlers ensure exceptions not propogated

(c) Exceptions can be raised (circumstances?)

Subtypes of floating point types
(named subtypes or on individual objects)

- Range constraint with dynamic values for the bounds
  (run-time checking)
- Accuracy constraint digits N (<= F'DIGITS)

  Mantissa length of model numbers reduced
  Exponent range the same
  Hence
  SF'MANTISSA <= F'MANTISSA
  SF'SMALL = F'SMALL
  SF'EMAX = F'EMAX etc
  SF'EPSILON >= F'EPSILON
- Implementations are unlikely to make use of restricted accuracies for subtypes

Exercises

Given two floating point types such that

F'DIGITS = 2 * G'DIGITS

does F'MANTISSA = 2 * G'MANTISSA ?

What is the largest X such that X does not overflow nor does 1.0/X underflow?

(Answers at back of notes)

Predefined Floating Point Operations

X, Y : FLOAT

+ X	no operation
- X	change sign
X + Y	addition
X - Y	subtraction
X * Y	multiplication
X / Y	division
ABS(X)	absolute value

(I : INTEGER

X ** I	exponentiate)

All these operations follow standard rules for bounding errors

Example 1 X, Y : F ; F'DIGITS = 5

X / Y with X = 15.0 and Y = 3.0

Both values are model numbers as is the mathematical result, in consequence, the machine result is exactly 3.0

Example 2

X * Y with X = 0. and Y = 10.0

X in model interval 16#0.19999# .. 16#0.1999A#

Y is a model number

Step 2 gives 16#0.FFFFA# .. 16#1.00004#

Step 3 gives 16#0.FFFF8# .. 16#0.10001#E1

Example 3

X + Y X = 1.0 Y = F'SMALL

Result in model interval 1.0 .. 1.0 + F'EPSILON

(as F'EPSILON > F'SMALL)

Exponentiate operator

(any floating point expression) ** (any integer expression)

Defined by repeated multiplication

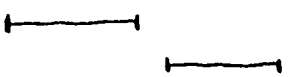
X **2 equivalent to X *X

X **(-2) equivalent to 1.0/(X *X)

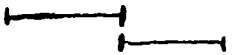
X **0 equivalent to 1.0

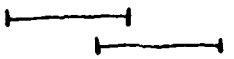
Relational Operators

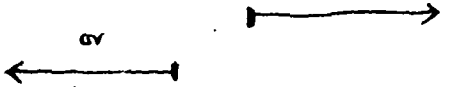
Result depends upon relationship between model intervals of each operand

(a)  correct result

(b) correct result

(c)  correct result or X op X

(d)  implementation defined

(e) 

implementation defined or NUMERIC_ERROR

Exercises

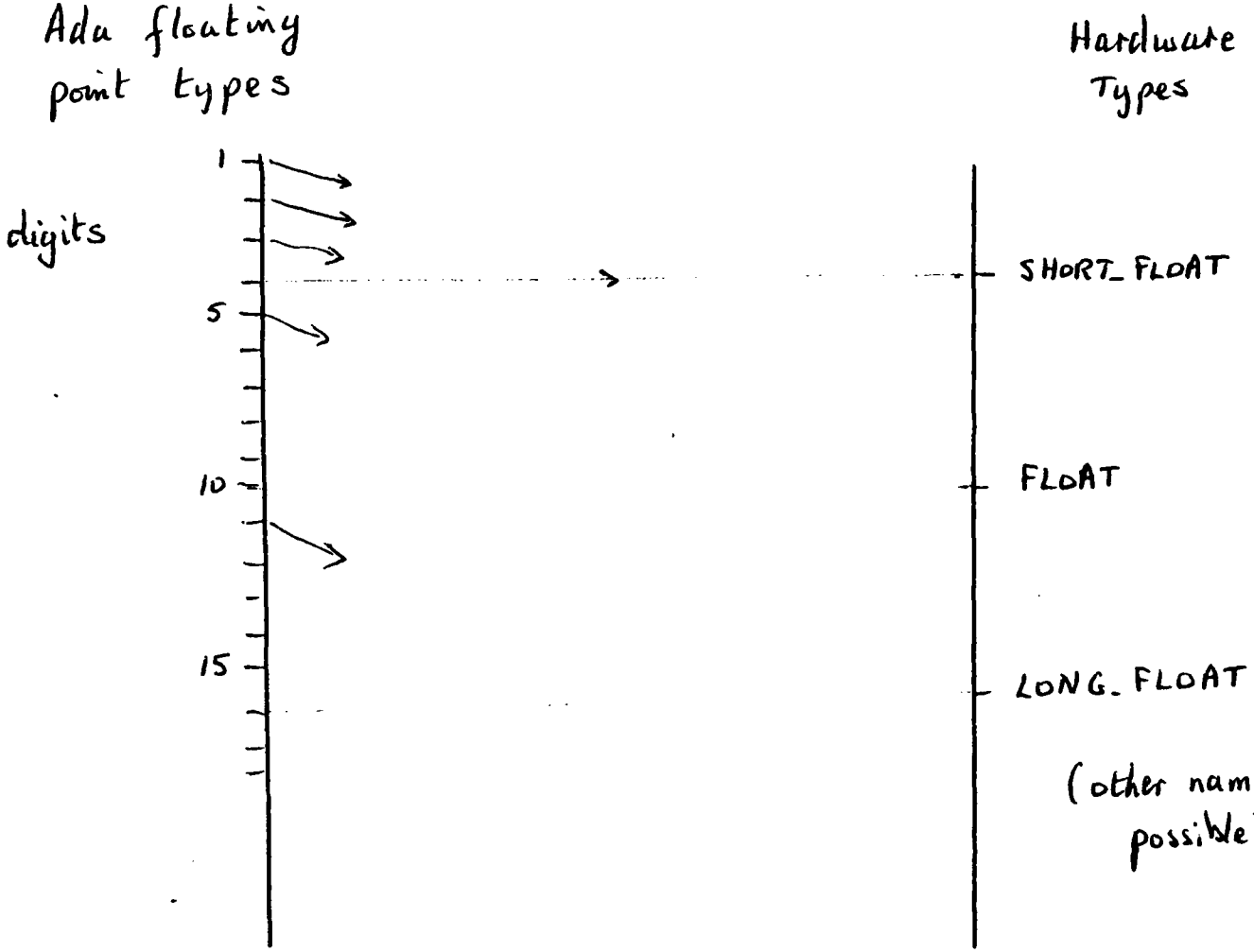
A, B, C : FLOAT;

Does $(A+B)+C = A+(B+C)$?

$A+B = B+A$?

What is wrong with the following?

```
A + 12
24 * B
C ** 2.0
```



Mapping used will depend upon implementation

type F is digits D;

shorthand for

type F is new FLOAT digits D;

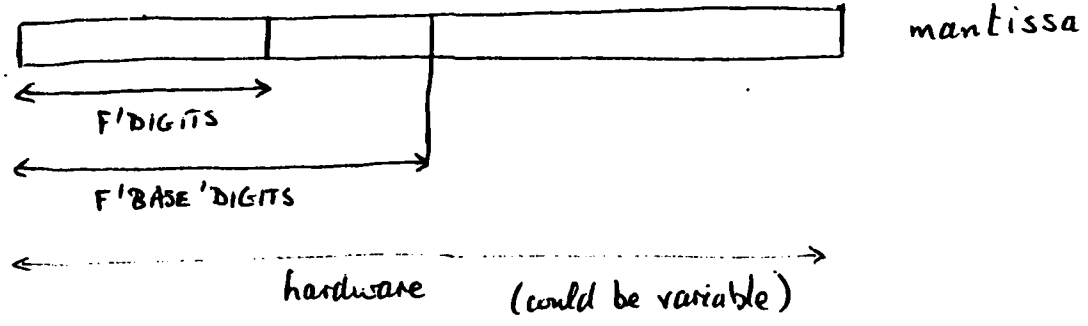
↑ appropriate hardware type

F is said to be *derived* from FLOAT.

Hardware type is 'BASE, which can be used in attributes

i.e. FLOAT'DIGITS = F'BASE'DIGITS

of course F'BASE'DIGITS >= F'DIGITS



Fixed Point Data Types

type Fx is delta D range L .. U;

- D — FX'DELTA
- L — FX'FIRST
- U — FX'LAST

Model numbers are integer multiples of FX'ACTUAL_DELTA.

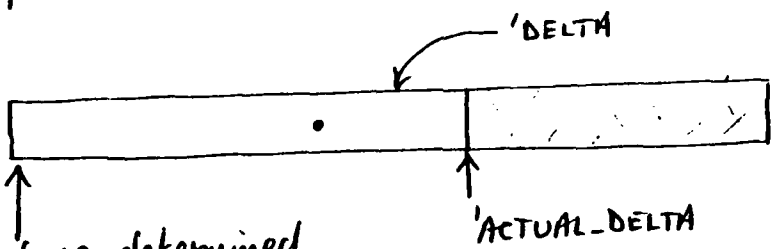
FX'ACTUAL_DELTA <= FX'DELTA

Sign * multiple * FX'ACTUAL_DELTA

0 <= multiple <= $2^N - 1$ (for some N)

A representation specification allows the programmer to specify the ACTUAL_DELTA.

Representation used for fixed point



Unlike floating point, a representation specification can remove the extra accuracy of the hardware.

```
type INTENSITY is delta 1.0/128 range 0.0 .. 127.0/128;
for INTENSITY'ACTUAL_DELTA use INTENSITY'DELTA;
for INTENSITY'SIZE use 7;
```

Not all model numbers can be represented as stored values because of a range constraint

Attributes of Fixed Point Types

F'DELTA - real literal value in type/subtype declaration

F'ACTUAL_DELTA - real literal value of the type/subtype F

F'BITS - integer value, the number of bits in an unsigned model number

F'LARGE - largest model number of type F

= (2 ** F'BITS - 1) * F'ACTUAL_DELTA

Accuracy constraints can be applied to fixed point types to give subtypes.

Implementation can, but need not, have different representation for subtypes

Exercises

```
type FX is delta D range L .. U;
```

(a) Are D, L, U model numbers?

(b) Can the range constraint be omitted?

(c) If L < 1.0 < U, is 1.0 a model number?

What is wrong with:

(d) `type FD is delta 0.01 range 0.0 .. SQRT(2.0);`

(e) `type FE is delta 0.01 range 0 .. 10;`

(f) `type FF is delta 10.0 range 0.0 .. 100 * FF'DELTA;`

Fixed Point operations without rescaling

X, Y : FX;

result type FX

I : any integer type

+ X	no operation
- X	change sign
X + Y	addition
X - Y	subtraction
I * X	equivalent to repeated addition
X * I	equivalent to repeated addition
ABS(X)	absolute value
X / I	division without rescaling

Literal values converted to within 'ACTUAL_DELTA of true value

Examples of error bounds on computations

(F'ACTUAL_DELTA = 1.0/128)

```
X : F := 10.1 ; --bounded by 1292.0/128 .. 1293.0/128
Y : F := X/2 ; -- bounded by 646.0/128 .. 647.0/128
Z : F := 2*X; -- bounded by 2584.0/128 .. 2586.0/128
W : F := 10.1 + X ;  -- same bounds as Z.
```

Conversion of literal values outside the range of the implemented type will raise CONSTRAINT_ERROR.

Operations giving values outside the range of the implemented type will raise NUMERIC_ERROR.

Fixed Point rescaling operations

Scale needed for result of multiplication and division cannot be determined statically.

The programmer is required to state the scale (type) of the result.

```
G1 := G ( F1 * F2 );
```

Result of F1 * F2 (or 6.48 * F1, F1/8.6, F2/F1 etc)
is Universal Fixed.

Arbitrarily high accuracy of Universal Fixed.

G reduces the accuracy to explicit type

Composite operation obeys the usual rules of the accuracy of real operations

What is the type of a numeric literal ?

Real literal - type Universal Real

Integer literal - type Universal Integer

Implicitly converted to type required by context

Literal expressions are expressions whose only operands are numeric literals, operations are the predefined ones.

Literal expressions are equivalent to writing the value instead.

```
I := 3_000_000 - 3_000_000;
     ^^^^^^^^^^^^^^^^^^^^^
     a literal expression = 0
```

Hence CONSTRAINT_ERROR will not be raised

Operations on Universal Integer are just the same as all integer types.

Universal Real encompasses capability of both float and fixed point.

```
ADD   : constant := 1.0 + 3.0;
SUB   : constant := 6.0 - 8.0;
PLUS  : constant := + 12.0;
NEG   : constant := - 8.4;
MULT1 : constant := 2 * 3.0;
MULT2 : constant := 4.0 * 10;
MULT3 : constant := 5.0 * 10.1;
DIV1  : constant := 10.0 / 2;
DIV2  : constant := 10.0 / 2.0;
EXP   : constant := 3.0 ** 2;
ABS1  : constant := ABS(6.0);
```

Relational operators as well

Careful with = and /=

Exercises

Are the following literal expressions?

(a) F'DIGITS + 10

(b) FX'DELTA / 10.0

(c) FX'LAST - 0.1

(d) 1 / FX'ACTUAL_DELTA

Which is more accurate (X: FX)

10.1 + 10.1 + X or 10.1 + X + 10.1

Use of Universal Real is both more accurate and more efficient (at run-time).

Blue's Algorithm for $\sqrt{\sum a_i^2}$

- Hard to avoid underflow/overflow
- Example of high quality algorithm
- Comparison with FORTRAN possible

Assume following context:

```
type REAL is digits D;
type VECTOR is array (INTEGER range <>) of REAL;
function SQRT (X: REAL) return REAL;
```

Specification of function:

```
function NORM (X: VECTOR) return REAL;
```

Article referenced should be studied for full details

Algorithm requires some values to be stored exactly.

This can be done by literal expressions in Ada

Examples

```
EB1 : constant := (REAL'EMAX + 1)/2;
        -- uses EMAX even and type Universal Integer

B1 : constant REAL := 2.0 ** (-EB1);  -- model number

OVERFL: constant REAL := REAL'LARGE * 52M;
        -- can be calculated by compiler

RELERR : constant REAL := SQRT (REAL (REAL'EPSILON));
        -- must be calculated dynamically
        -- conversion necessary
```

The vector could be too big

```
N : constant INTEGER := X'LENGTH;

if N > 2 ** REAL'MANTISSA then
      raise CONSTRAINT_ERROR;
end if;
```

However, if INTEGER'LAST < 2 ** REAL'MANTISSA, the comparison will yield CONSTRAINT_ERROR directly (which it must not). However, in this case, the comparison is unnecessary since the condition is always FALSE.

Algorithm accumulates three separate sums of squares for the small, medium and large valued terms.

Afterwards, the sums are inspected to give the combined result

```
return ABIG * SQRT (1.0 + (ASML/ABIG) **2);
```

This calculates

```
SQRT ( ABIG **2 + ASML **2)
```

without unnecessary risk of overflow.

Instead of using REAL attributes, the algorithm could use the attributes of the hardware type is.

REAL'LARGE becomes REAL'BASE'LARGE

Advantages

- Closer match to actual hardware capability
- Only one routine for each hardware type

Could also use machine-dependent attributes (outside model).

Fixed Point Approximation

Smooth functions can be approximated by power series

```
Y := A + B*X + C*X**2 + D*X**3;
```

This method of evaluation gives at least one extra rounding error on every additional term. Nested multiplication is better:

```
Y := ((D*X+C)*X+B)*X+A;
```

Algorithm is taken from Cody and Waite - see references.

Given FR : purely fractional fixed point type

```
G : FR ;  -- reduced argument for sine/cosine
Y : FLOAT;
```

Approximation:

```
Y := FLOAT(
        FR((
          FR((
            FR((
              FR(0.00066_60872 * G)
            - 0.01267_67480) * G)
          + 0.13332_84022) * G)
        - 0.66666_62674) * G)
        );
```

```
package SIN_COS is
   function SIN ( X: FLOAT) return FLOAT;
   function COS ( X: FLOAT) return FLOAT;
end SIN_COS;
```

Use of FLOAT just an illustration.

Algorithm notes sign, add $\pi/2$ for COS, then calls COMMON_PART.

COMMON_PART does

1) check argument not too large
2) performs range reduction
3) does fixed point approximation
4) converts back and changes sign if necessary.

Integer type used for argument reduction:

```
B: constant := FLOAT'MANTISSA;
type INT is range 0 .. 4 * 2 ** (B/2);
```

Examples of number declarations:

```
PI : constant := 3.14159_26535_89793_23846;
PI_DIV_2 : constant := PI/2;
ONE_DIV_PI : constant := 1.0/PI;
```

(use of number declarations preserves accuracy)

Argument reduction is difficult

To do

$$F := X - \text{Integer\_part\_of}(X/\pi) * \pi$$

without loosing unnecessary accuracy

This is done by

```
F := ((X1 - XN * C1) + X2) - XN * C2;
```

where C1 + C2 = π to more than machine accuracy

```
C1 : constant := 8#3.1104#;
C2 : constant := -8.9089_10206_76153_73566_17E-6;
```

Conditional Compilation

Sizes of mantissa of FLOAT determines the number of terms needed in fixed point approximation. Rely upon compiler optimization to give required code. If FLOAT is too inaccurate, raise CONSTRAINT_ERROR.

```
if B <= 24 then
   --
elsif B <= 32 then
   --
elsif B <= 50 then
   --
elsif B <= 60 then
   --
else
   raise CONSTRAINT_ERROR;
end if;
```

(warning from compiler if CONSTRAINT_ERROR always raised)

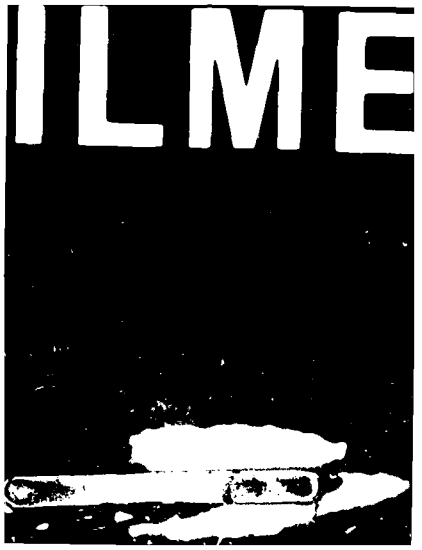
Example of generics

- For any real type
- Exploit generality of text
- Appropriate for general purpose routines

```
generic
    type REAL is digits <>;
package COMPLEX_OPS is
    type COMPLEX is record
                        RE, IM : REAL;
                        end record;
    function "-" (X : COMPLEX) return COMPLEX;
    function ABS (X : COMPLEX) return REAL;
    function "+" (X, Y : COMPLEX) return COMPLEX;
    function "-" (X, Y : COMPLEX) return COMPLEX;
    function "*" (X, Y : COMPLEX) return COMPLEX;
    function "/" (X, Y : COMPLEX) return COMPLEX;
end COMPLEX_OPS;
```

User's Program

```
type R is digits 10;
package MY_C is new COMPLEX_OPS(R);
use MY_C;
U, V : COMPLEX := (1.0, 2.0);
W : COMPLEX :=  U * V;  -- COMPLEX multiplication
```


ILME